TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Khuldoon Bukhari 177331IASM

AGILE EMBEDDED SOFTWARE DEVELOPMENT FOR AN ARM CORTEX-M BASED IOT DEVICE

Master's Thesis

Supervisor: Eduard Petlenkov
PhD

Tallinn 2019

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Khuldoon Bukhari 177331IASM

AGIILSE SARDSÜSTEEMI ARENDAMINE CORTEX-M TÜÜPI ASJADE INTERNETI SEADMELE

Magistritöö

Juhendaja: Eduard Petlenkov
PhD

Tallinn 2019

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Khuldoon Bukhari

25.04.2019

Abstract

The thesis focuses on making the process of embedded software development for an IoT (Internet of things) enabled device more agile by introducing some abstraction layers. The software development models, as well as the software architectures in embedded Cortex-M based methods, are extensively discussed and compared to the developed device. Based on the analysis using a real-time operating system and having the means to deploy software dynamically makes the development process more agile.

This thesis is written in English and is 31 pages long, including 7 chapters, 14 figures and 1 table.

Table of contents

List of abbreviations and terms

2G	Second-Generation Cellular Technology
3G	Third-Generation Cellular Technology
ADC	Analog-to-Digital Converter
ANT	Low-Power Multicast Wireless Sensor Network Technology
API	Application Programming Interface
ARM	Advanced RISC Machines
BLE	Bluetooth Low Energy
CMSIS	Cortex Microcontroller Software Interface Standard
CPU	Central Processing Unit
DAP	Debug Access Port
DC	Direct Current
DFU	Device Firmware Update
FPU	Floating Point Unit
GLONASS	Global Navigation Satellite System
GNSS	Global Satellite Positioning System
GPIO	General-Purpose Input/output
GPS	Global Positioning System
HAL	Hardware Abstraction Library
I/O	Input/output
I2C	Inter-Integrated Circuit
IC	Integrated Circuit
IMU	Inertial Measurement Unit
IoT	Internet of Things
ISR	Interrupt Service Routine
IT	Information Technology
LED	Light Emitting Diode
MAC	Multiply-Accumulate
MCU	Microcontroller Unit
MPU	Memory Protection Unit
MQTT	Message Queuing Telemetry Transport
NVIC	Nested Vector Interrupt Controller
OS	Operating System
OTA	Over the Air

List of abbreviations and terms

PCU	Power Control Unit
RGB	Red, Green, Blue
RISC	Reduced Instruction Set Computing
RTOS	Real-Time Operating System
SCB	System Control Block
SHA256	256-Bit Secure Hashing Algorithm
SIMD	Single Instruction, Multiple Data
SoC	System on Chip
SPI	Serial Peripheral Interface
SPL	Standard Peripheral Library
SVD	System View Description
SysTick	System Tick Timer
TCP/IP	Transmission Control Protocol/Internet Protocol
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus
WFI	Wait-For-Interrupt
XML	extensible Mark-up Language
XP	extreme Programming

List of figures

List of tables

1 Introduction

In 2008, there were around 30 embedded microprocessors per person in developed countries with at least 2.5 million function points of embedded software. The worldwide market for embedded systems was around 160 billion euros, with an annual growth of 9 percent [1]. According to some estimates embedded microprocessors account for more than 98 percent of all produced microprocessors, thus vastly surpassing computing power in the IT industry. According to ARM®, its partners alone shipped over 22 billion Cortex-M based devices of 32-bit embedded processors in just ten years [2].

Connectivity has become a critical feature given the growing interest for IoT; everyday objects are becoming smart, connected and automated. It is only natural that these smart devices require security and device management, thus further increasing the complexity of the embedded software. The increasing difficulty of the projects and growing competition make the technical and commercial environment of embedded software more intense.

Conventional methods of embedded software development recognize quality, cost, and development time as points on a triangle; any two get optimized at the cost of the third [3]. Considering the commercial and industrial stakeholders working in this field, the strategy is seriously flawed, the quality, cost and time matrix needs a significant improvement. To do so will require disruptive changes in the way we currently design, develop, and test embedded systems [3]. Perhaps, the embedded software developers can get some inspiration from the general software industry to deal with a lack of agility and scalability in the embedded realm.

1.1 Background

A significant portion of the work done related to this dissertation has been done to facilitate or improve the embedded software (also known as firmware) development process at the author's place of employment. The author works as an embedded software developer in the CompanyX where his primary responsibility is to develop embedded software for DeviceX. DeviceX is an IoT enabled ARM Cortex-M4 based device that acts

an anti-theft accessory for bicycles with some very advanced tracking features. The device is described with additional details in the chapter 4.

1.2 Aim and motivation

The task was to develop embedded software/firmware for DeviceX according to the customer specifications. The hardware design of the DeviceX was already complete at the time the author took over the project. `The firmware, however, only validated the hardware prototype and served as proof of concept for the customer. Therefore, the aim of the development became to port the existing firmware to the newly finalized device hardware and later to implement the agreed-upon specifications of the product.

There were some critical differences in the initial hardware prototype and the finalized hardware including change of the primary microcontroller and addition of some new peripherals. The amendment resulted in a complete change in the pin mapping of the host MCU (Microcontroller Unit) and its peripherals. While the difference in the MCUs was not significant as both the MCUs followed the same ARMv7M architecture, it took a considerable amount of development effort to port the existing firmware to the new MCU.

After, the above-stated experience, along with some other adventures in the field of embedded software, the author chose to research the area that could make the embedded software/ firmware development more agile at the author's current place of occupation. However, due to the diversity and the versatility of the requirements from embedded software, the research topic was narrowed down to the software running on ARM Cortex-M based MCUs, and further only considers the applications that have non-safety critical, non-mission critical needs. This research also does not specifically focus on real-time application requirements, but does not omit it.

1.3 ARM Cortex-M

ARM is a family of computer processor architectures based on RISC (Reduced Instruction Set Computing). ARM Holdings develops the architectures and licenses it to the interested silicon manufacturers who then design their own SoC's (System on Chip) based on the ARM architectures. The reduced instruction set leads to a lower transistor count on the chip, which leads to a smaller integrated circuit (IC), lower cost and power

consumption. That is why ARM processors are widely used in consumer electronic devices such as tablets, smartphones, wearables and other deeper embedded systems [4].

The Cortex-M processor family is a low-end performance family of the arm processors and is generally used for generic microcontroller products. Nonetheless, these processors are still considerably powerful when compared to the processors of the most microcontrollers. The Cortex-M processor family comprises of different products to address various demands. Some processors from the Cortex-M family have maximum clock frequency going up to 400MHz [5]. However, performance is seldom the only determinant while choosing a processor. Often, power consumption and cost are the decisive factors as well.

Cortex-M processor products support different series of instruction set, for example, ARMv6-M architecture with the highest energy efficiency in the family is ample for general data processing and regular input/output (I/O) control tasks. The processors from the ARMv7-M architecture (such as the Cortex-M3 and Cortex-M4) provides additional instructions for hardware divide, bit field processing and Multiply-Accumulate (MAC) which accelerates data processing for more complex applications. The Cortex-M4 processor provides further instructions such as SIMD (Single instruction, multiple data) and floating-point unit (FPU) which provides optimized single precision floating calculation in the floating-point hardware.

The Cortex-M23 and Cortex-M33 processors are the newest members of the Cortex-M product family featuring ARMv8-M architecture. The processors offer similar performance to the Cortex-M0 and Cortex-M4 respectively with better power efficiency. They retain the tested embedded characteristics of the Cortex-M family while adding essential security foundations via the addition of TrustZone technology [5] [6]. The TrustZone technology allows the designers of a microcontroller to define the memory spaces into hardware-enforced secure and non-secure areas while also providing access to the secure information from the non-secure areas of the memory via predefined application programming interfaces (API) [6].

The Internet-connected embedded systems or IoT enabled embedded systems can be designed using different processor architectures. While some architectures may outperform others in specific domains, the low-cost ARM Cortex-M processors can

feature multiple connectivity channels as well as the corresponding software stacks and therefore meet the demands of many IoT applications. Enabling developers to develop high-performance feature-rich products [2].

2 Software development processes

According to [7], the development processes best suited for the embedded software/firmware are the sequential or linear processes such as Waterfall or V-model. In both Waterfall and V-models, the requirements are defined before the development begins, i.e. the full specification is required at the beginning of the project. Sequential processes are therefore preferred in mission or safety-critical systems, but for non-critical software, agile processes can offer the advantage of short development time. The following sections will describe the development models briefly.

2.1 Waterfall development model

Waterfall model is a form of a sequential development model where a sequence of stages is followed until the conclusion of the project. The output of each stage becomes the input for the next, Figure 1. Waterfall model requires the requirements to be defined before going to the next phase, i.e. the previous step is always frozen before the next step. This results in the testing phase to be carried out only when the software is fully developed, which most of the times mean the defects are found very late in the development life cycle. [8]

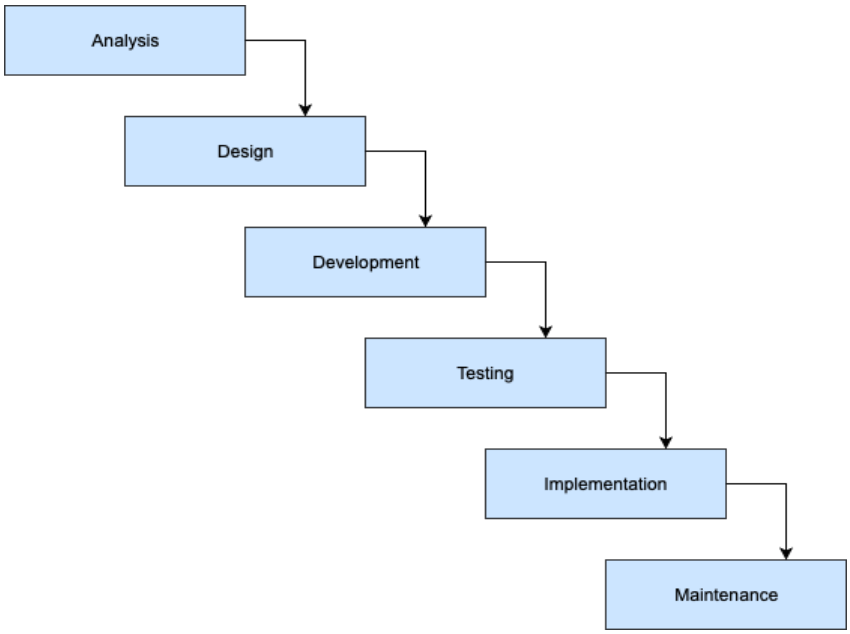

Figure 1. Waterfall model [8]

The waterfall model being a linear model is relatively easy to implement and requires minimal resources to execute. However, any changes in the requirements mean not being implemented in the ongoing development cycle [8].

2.2 V-development model

“The V model (Validation & Verification model) is a modified version of the Waterfall method” [8]. Unlike the Waterfall method, the V model does not follow a linear axis; instead, the stages turn back upwards after the analysis and design phase is completed; representing the letter "V" Figure 2. The developers and testers work parallelly and thus form a relationship between each phase. The main disadvantage here is the resource-intensive nature of the model. It requires much documentation like functional specifications for high-level design, low-level design, unit testing, system testing, integration, and testing. Therefore, it is one of the most rigid development models as any change in the requirements results in the need for documentation to be updated [8].

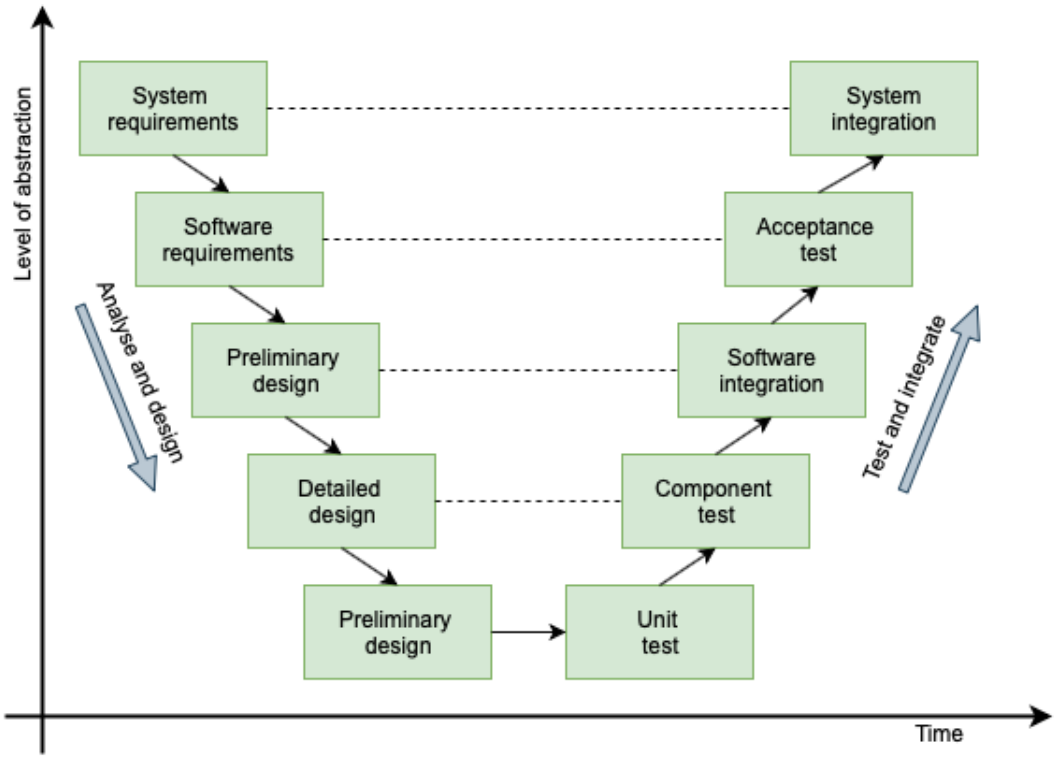

Figure 2. V model representation [9]

2.3 Agile development

Agile software development is a collection of software development methodologies/frameworks based on iterative development, where requirements and solutions evolve through collaboration between functional teams. Agile processes allow rapid delivery of high-quality software and align development with customer needs. The focus of agile development is customer satisfaction achieved through rapid and continuous delivery of small and serviceable software. The main advantage of the agile model, however, is the ability to respond to the changing requirements of the project even in the later stages of development. The agile model can be best represented by Figure 3.

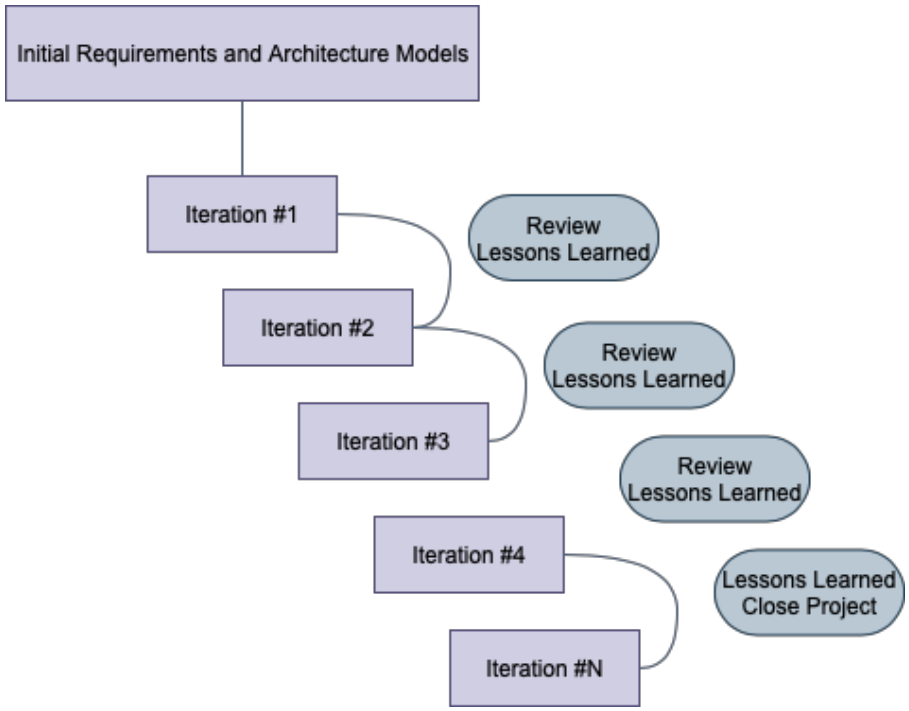

Figure 3. Representation of the agile processes [8]

One of the subsets of Agile is Scrum; it is the most used lightweight process framework for agile development. It is often used to develop complex software and product development, using iterative and incremental practices [10]. Scrum significantly increases productivity and reduces time to market compared to the sequential models. The software is delivered frequently with continuous inputs from the users. Moreover, scrum processes allow organizations to adjust smoothly to rapidly-changing requirements and produce a product that meets evolving system needs. Some disadvantages should be taken into consideration while using the Scrum development model. For example, in case of smaller projects, using the agile model is certainly profitable, but if it is a large project,

then it becomes difficult to judge the efforts and the time required for the project in the software development life cycle [8].

Another subset of agile development is extreme programming (XP); the methodology describes a "lightweight methodology for small-to-medium-sized teams developing software in the face of vague and rapidly changing requirements" [11]. XP builds twelve established practices for software development and takes them to extreme levels. All the practices are wrapped around the core values [11]:

- Communication: promotes better communication between the team through pair-programming and coding standard.
- Simplicity: the core belief here is, it is better to do a simple working thing today and pay a little extra in the future to change it than it is to make a complicated, reusable thing and pay for it today, but maybe never use it in the future.
- Feedback: the product is put to production in small releases as soon as possible to get feedback. Collective code ownership and continuous integration furthermore help to get immediate feedback.
- Courage: programmers should have the courage to refactor even when it means a sizeable architectural change. They should have the courage to throw away harmful code, even if it works.

Sequential development models have been used so far in the development of the DeviceX. In the context of this thesis, the biggest motivation for using the agile software development method for the development of DeviceX is to coupe up with the changing requirements and still able to develop further with least efforts. Hence, to make agile development work, it is essential for the software to be least susceptible to hardware changes in case the hardware requirements change. The software should also stay maintainable even after the incremental software updates that increase the complexity. The next chapter focuses on the software standards and its effects on the development.

3 Cortex microcontroller software interface standard (CMSIS)

The standardization of the software infrastructure is quintessential to ensure software compatibility between various development tools and different software solutions. The rise in the complexity of the embedded systems has resulted in higher software development efforts from the developers as well as the use of third-party software solutions. A modern-day embedded software project might include software components from many different sources, for example [12]:

- Software developed by in house developers.
- Software reused from previous projects.
- Device-driver libraries from microcontroller vendors.
- Embedded operating systems.
- Communication protocol stacks (e.g. TCP/IP, Bluetooth) from third party vendors.

Consequently, CMSIS was developed by ARM to allow microcontroller and software vendors to use consistent software architecture to develop software solutions for Cortex-M microcontrollers.

The purpose of CMSIS is to enhance the software reusability and software compatibility of the Cortex-M processors along with implementing a toolchain agnostic and open platform for developers. CMSIS makes it easier to reuse code among Cortex-M projects that minimize the time to market and verification efforts. Additionally, software source diversity is made possible by having a consistent API for processor core access functions, system initialization method, and unified style for defining peripherals that reduces the risks in integration. CMSIS compliant device drivers can be used with multiple compilation tools, providing much greater freedom while also being open source. Since the inception of the CMSIS project, it has intensively diversified and the one mentioned previously is now called CMSIS-Core. There are also other CMSIS projects that have matured over the years like CMSIS-SVD, CMSIS-RTOS, CMSIS-DAP, etc [12]:

- CMSIS-SVD: the CMSIS System View Description is an XML based file format to describe peripheral set in the microcontroller products. Debug tool vendors can then use the CMSIS SVD files prepared by the microcontroller vendors to construct peripheral viewers quickly.
- CMSIS-RTOS: the CMSIS-RTOS is an API specification for the embedded OS running on Cortex-M microcontrollers. This allows middleware and application code to be developed for multiple embedded OS platforms and allows better reusability and portability.
- CMSIS-DAP - the CMSIS-DAP (Debug Access Pori) is a reference design for a debug interface adaptor, which supports USB to Serial protocol conversions. It allows low-cost debug adaptors to be developed which work with multiple development toolchains.

3.1 CMSIS-Core

The CMSIS core is vital when it comes to the software development for Cortex-M based microcontrollers. It standardizes as well as provides access to the following sections of the controller [12]:

- Processor peripherals definitions, Incorporating the registers of the Nested Vector Interrupt Controller (NVIC), system tick timer (SysTick), Memory Protection Unit (MPU), various programmable registers in the System Control Block (SCB), and some debugging related registers.
- API for various functions for interrupt control using NVIC and functions for accessing special registers in the processor that accommodate software portability.
- API for accessing processors special instructions for special purposes, e.g., Wait-For-Interrupt (WFI), power management that cannot be generated by generic C language. CMSIS core implements a set of functions to allow these instructions to be accessed with C language. Without these functions, the developers would have to rely on toolchain specific solutions such as intrinsic functions or assembly to insert special instructions into the application, which make the software less reusable and requires in-depth knowledge of the toolchain in order to handle them correctly.

- Definitions of system exception handlers give the corresponding system exception handlers standardized names, and it makes it much simpler to develop software solutions that can be applied to various Cortex-M products. It is especially essential for embedded OS developers, as the embedded OS demands the use of numerous types of system exception.
- System initialization API for the configuration of clock circuitry and power management registers before the application starts. In CMSIS-compliant device-driver libraries, these configurations are placed in a function called "Systemlnit()". The implementation of the function varies but having a standardized function name, a standardized way that this function is used and a standardized location where this function can be found makes it much easier for a designer.

3.2 Structure of CMSIS-Core

The microcontroller vendors manufacturing Cortex-M devices provide the device-driver library packages. Some amount of the files from the device-driver library are developed by ARM and are common to various microcontroller vendors. Other files are vendor or device specific. Nevertheless, CMSIS can be defined in the following layers [12]:

- Core Peripheral Access Layer: it contains name definitions, address definitions, and helper functions to access core registers and core peripherals. This layer is processor specific and is provided by ARM.
- Device Peripheral Access Layer: it contains name definitions, address definitions of the peripheral registers, as well as system implementations including interrupt assignments, exception vector definitions. The device files can be identical if the product is from the same vendor.
- Access functions for Peripherals: This layer is vendor specific for the driver code for peripheral accesses and is an optional layer. The developer gets to choose whether to develop an application using the peripheral driver code provided by the vendor, or directly program the peripherals.
- Middleware Access Layer: this layer is under development. The goal is to develop a set of APIs for interfacing common peripherals such as UART, SPI, and Ethernet. This layer will then allow middleware developers to develop

applications based on this layer to enable the software to be ported between devices easily.

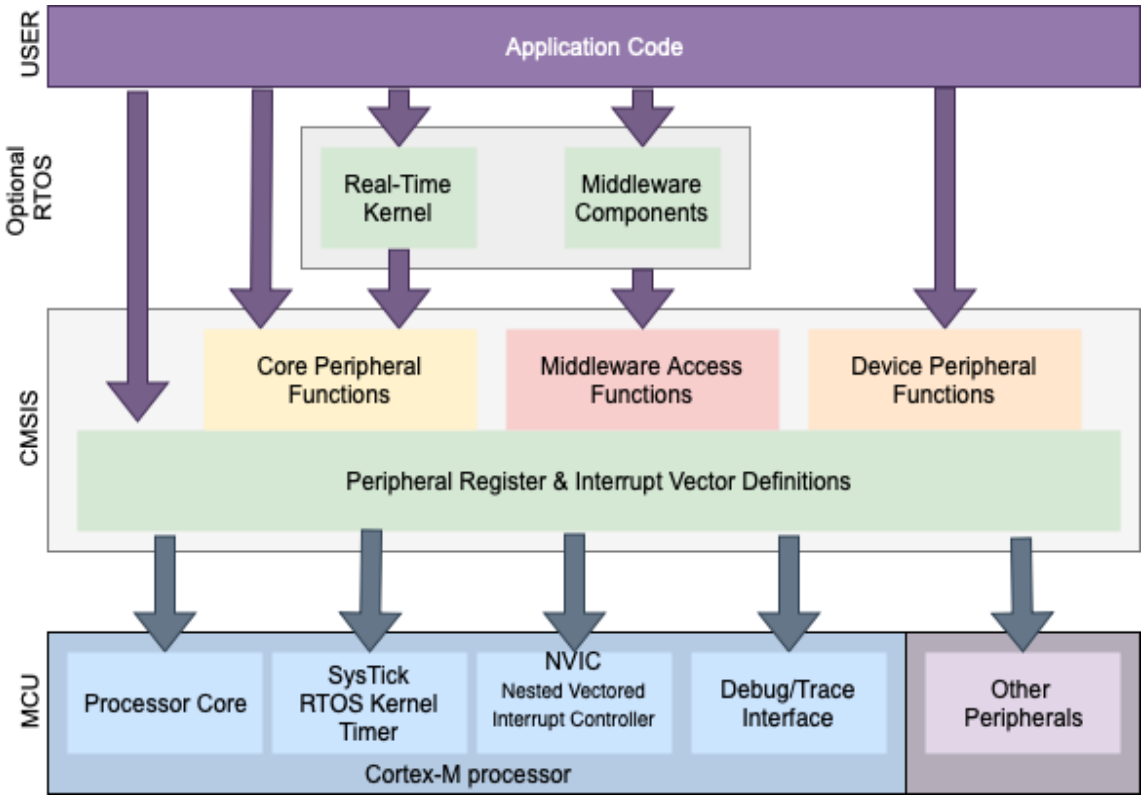

Figure 4. The structure of CMSIS-M Core [12]

The roles of the various layers are summarized in the figure above.

3.3 Shortcomings of CMSIS

Even though the CMSIS is supposed to be a vendor agnostic hardware abstraction layer for the Cortex-M series, in reality, the vendors do not always comply completely with ARM standards. Often the vendors choose to ignore the APIs and continue to provide their implementations instead. Furthermore, it takes a substantial amount of time for the vendors to adopt the newly developed layers causing the application developers to opt for the vendor-specific older implementations that make porting to different hardware or an RTOS hideous. ST electronics, for example, has a wide variety of Cortex-M based MCUs on the market and offer two kinds of device libraries; the standard peripheral library (SPL) and the hardware abstraction library (HAL) as CMSIS compliant option. Instead of newer

products supporting only CMSIS compliant version they continue to develop SPL with only partial CMSIS compatibility.

4 DeviceX

The DeviceX is an anti-theft/tracking device for the bicycles that collects data from the surrounding sensors and uploads it to a remote server using mobile communication networks. Additionally, the device communicates with a companion mobile app as a user interface over Bluetooth. The device is capable of recording data from cadence sensor, speed sensor, heart-rate sensor, etc over the ANT network.

4.1 Hardware description of the deviceX

The deviceX is a modular embedded system on a board design featuring an ARM Cortex-M4 based 32-bit microcontroller as its central processor. The processor interacts with its external modules using various communication protocols such as SPI, I2C, and UART. The device also features Bluetooth low energy (BLE) and ANT wireless connectivity. The hardware interactions of the BLE and ANT+ is not described as doing so will disclose the company's intellectual property. The figure 5 shows the block diagram of the system and the following section cover the functionality of each peripheral in a clockwise direction:

- The cellular connectivity module is a gateway for the device to connect to mobile networks such as 2G and 3G. The module communicates with the host MCU using the UART protocol. All the communication between the remote server and the deviceX happens through this module.
- External flash storage provides additional data storage to the deviceX and is accessible through an SPI bus.
- The global satellite positioning system (GNSS) module acts as low power GPS, GLONASS receiver that helps to determine the location of the deviceX. The module also communicates with the MCU using SPI.
- The RGB (red, blue, green) Light emitting diode (LED) acts as an indicator for the intended functioning of the deviceX. The LEDs are connected to the three individual GPIOs of the MCU.

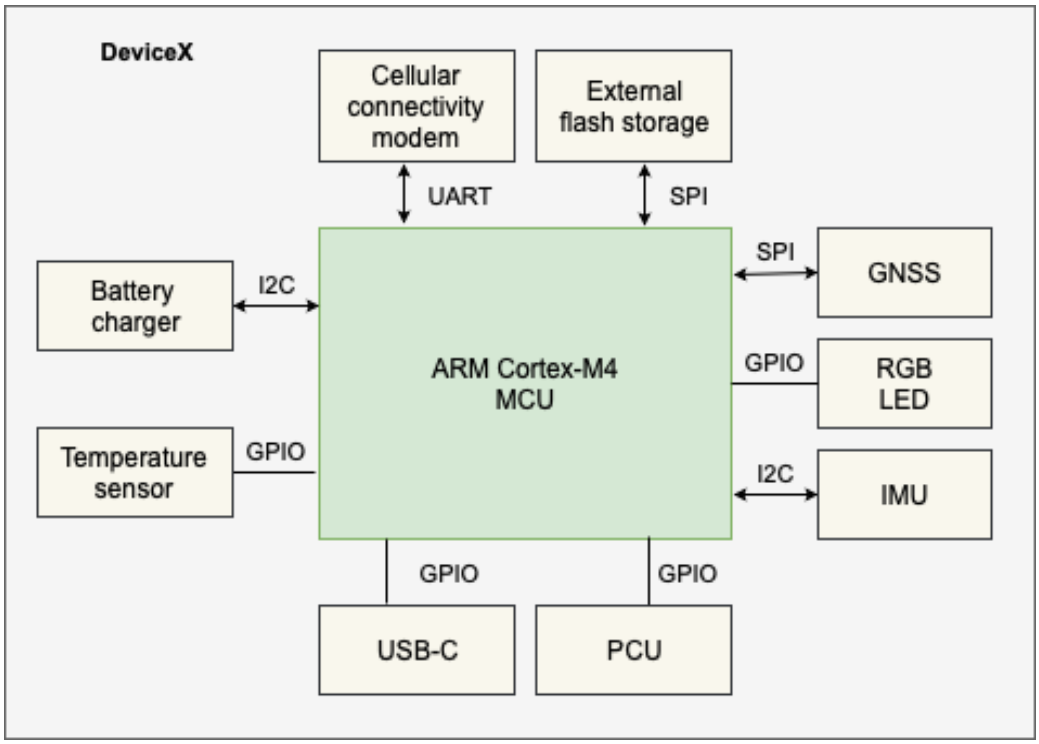

Figure 5. Block diagram of the deviceX

- The Inertial measurement unit comprises of accelerometer and gyroscope. The accelerometer provides the G-force measurement. The gyroscope helps to determine the orientation of the deviceX in 3-dimensional space. The IMU also has a particular output that connects to one the GPIOs of the MCU. This output can produce 3.3 V at the set acceleration value.
- The power control unit contains the circuitry to control the power to each peripheral module. Each switch or transistor connects to a GPIO on the MCU allowing to turn off any peripheral that is not required at that moment. Additionally, it holds the circuitry to measure different voltage levels on the board along with the input voltage to the charging module as well as the output of the charging module and the output of the battery. The outputs from this part of the PSU are connected to the different GPIOs with analogue to digital converter (ADC) channels.
- The USB-C in this device is only used for charging the battery of the deviceX. However, the CC1 and CC2 lines of the USB-C connect to the two ADC channels of the MCU that are used to calculate the maximum output current of the external charger.

- The temperature sensor is a thermistor that is connected to the ADC channel of the MCU and is used calculates the battery temperature.
- The battery charger module contains a DC-DC converter as well as some other power-related functions. It also allows changing the charging current of the battery as per the MCU instructions through an I2C bus.

4.2 Software requirements

The software requirements for the DeviceX were designed to comply with the customer specification. The functional requirements are essential as the software is always validated against the requirements. The requirements for the software/firmware are as follows:

- The device should be able to connect the remote server using cellular networks 2G or 3G depending on the availability of the networks. Additionally, the remote server should be able to receive data from the device as well send commands to the device.
- The device should be able to advertise itself over BLE and connect to a BLE central device. Also, sending data to the connected device and receiving commands from the same device should be possible.
- The device should scan for the supported ANT+ sensors in the range whenever it is active and receive and parse the data transmitted by the sensor(s). The parsed data should be sent to the connected BLE central device. The same sensor information should be sent to the remote server with the data collection rate set by the server.
- The device should be able to collect location information from the GNSS module onboard, parse the received data and send it to the remote server in the pre-specified format.
- All the data collected from all the sources should be saved in the external flash memory (in a specified format) until it is sent to the server.
- The device should be able to detect the motion of the bike and notify the server if the bicycle was in a locked state.

- The device should detect the idle state of the bicycle and conserve power by going into a sleep mode. Additionally, should send a status notification about the sleep mode to the server.
- The device should calculate the state of charge of the battery and convey it to the server as well as the BLE central device.
- The device should Initiate charging when a proper power source is connected with the maximum current rating of the source. Also, the charging state should be indicated on the status LED. The charging should stop when the battery is full or ambient temperature is not in the specified range and indicate it on the status LED.
- The device should be able to update its software through the internet or BLE.

4.3 Flash memory map

Flash memory is an electronically programmable and erasable non-volatile computer storage medium. The term non-volatile means the data is retained in the memory when the power source is absent. Cortex-M4 based MCUs usually have on-chip flash memory, for storing the application and application-related data.

The DeviceX has a 512kB on-chip flash memory; the software uses the available space in the following ways, Figure 6:

- The first 180 kB are reserved for the vendor specific implementation of the CMSIS core as well as application stacks for various communication protocols including but not limited to BLE, ANT, UART, SPI, and I2C.
- The application utilizes the 138kB (shown as a green area in the figure 6) of the memory. This area holds the main application code that provides the all most all the functionality of the device. The next 138kB of the green space remains unused most the times as it is only used at the time of the application update. More details in the following sections.
- The application uses the next 4Kb for storing different types of application data like flags, user settings and most importantly the private key. A key pair is generated at the time of programming the device, and then the private key is written in the flash memory while as the public key gets stored in the server for authentication.

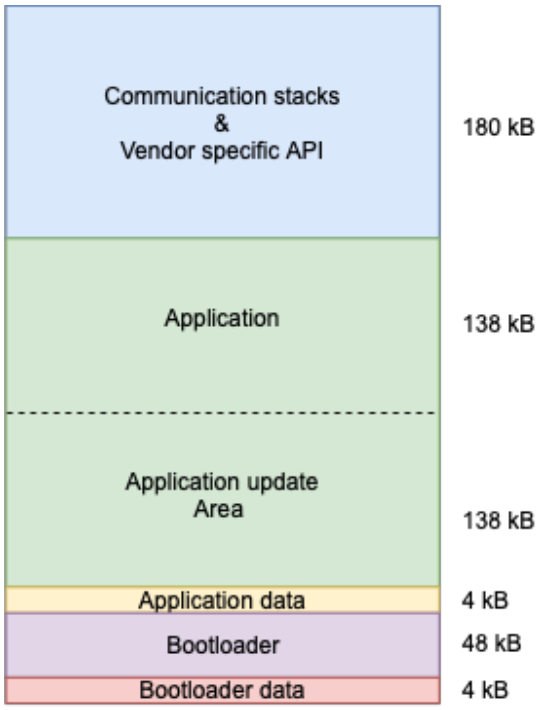

Figure 6. Memory map of the DeviceX

- The bootloader uses the next 48kb of the memory. The bootloader is just a piece of software that runs before the application to check the validity of the application and provides a means to update the device software/ firmware.
- The bootloader uses the next 4kB for storing its data including another key that is shared between all the devices and used exclusively for device updates only.

4.4 Bootloader architecture and flow

The bootloader of the DeviceX is a custom software designed to update as well as validate the main application. It is the first piece of software that gets executed upon system boot-up or a reset. BLE acts the transport layer for the device firmware update (DFU), and the bootloader manages the whole process. The bootloader receives two files form the BLE central device; one contains the main application in .bin format and the second .dat file contains: a signature. The bootloader uses polling and interrupts hybrid architecture as shown in the figure 7.

As part of the system initialization, the bootloader first controls the power supply and manages the charger (if connected) and then later continues to check the DFU flag. Only, the main application can set the DFU flag through the BLE or server commands. In the presence of the DFU flag, the bootloader initializes a watchdog and the BLE stack.

Otherwise, it jumps to the main application without any further operations. A watchdog is countdown timer that upon completion causes a system-wide interrupt that results in ISR execution. In this case, the ISR triggers a system reset.

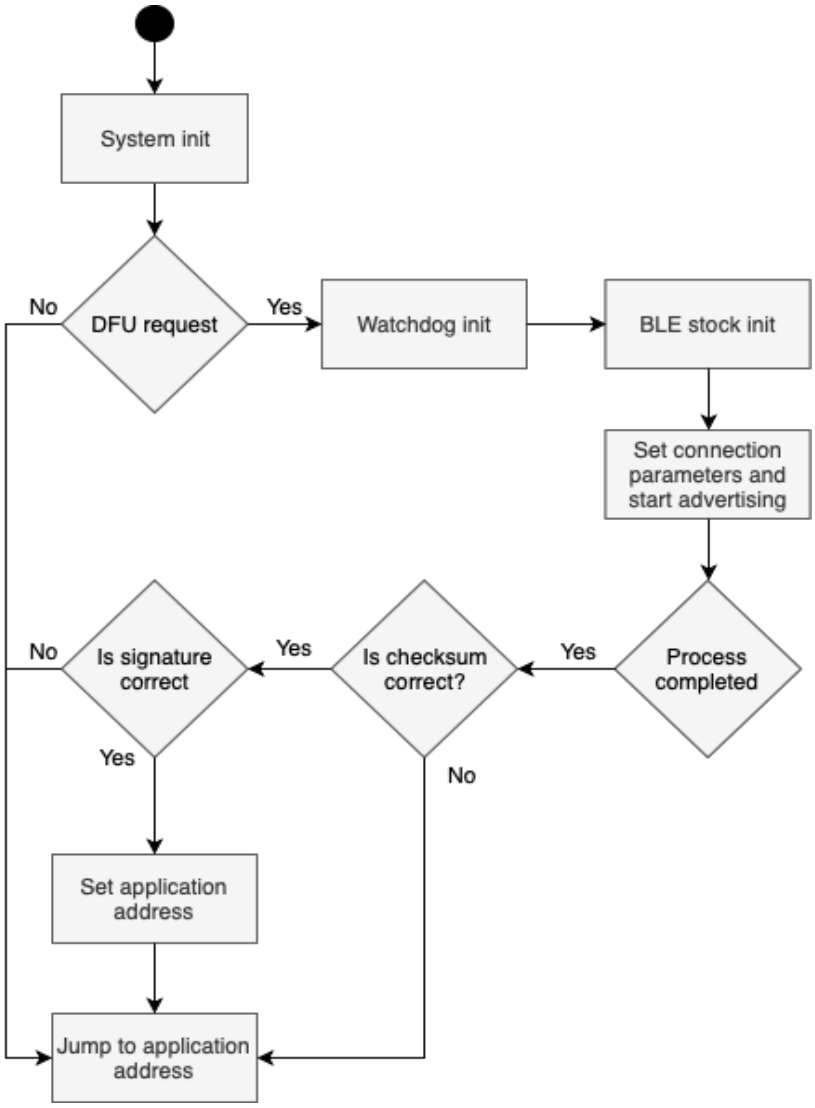

Figure 7. Bootloader software flow

After the BLE stack initializes, the bootloader sets the proper connection parameters and also starts advertising itself over the BLE. At this stage, the bootloader simply waits for the events, i.e., the interrupts from the BLE stack. Each BLE event has an associated handler that gets triggered by the corresponding event. If there are no events, the watchdog will interrupt at some point and cause a system reset. The details about BLE handling are omitted because of intellectual property infringements as well the device security concerns. In case a central BLE device is connected and starts sending the data packets corresponding handler copies the data to the flash memory and resets the

watchdog. This process continues until all the data is transferred and then the "process complete" flag is set by the handler.

Next, the bootloader validates the signature of the application, if the hashes match then the new address of the application gets stored in the master boot record and the old firmware or the application is deleted. Finally, the bootloader jumps to application and resumes routine operation.

4.4.1 Security

The bootloader can only be accessed by sending a command through BLE or the server. Only the main application can connect to the server to receive the command, so the application is not susceptible to any threat from that side; not at least within the context of this dissertation. However, any BLE central device can connect to the main application through BLE and theoretically send an update command provided the individual knows the exact message to write to the exact BLE characteristic of the correct BLE service. Then the application will jump to the bootloader and start advertising itself. Again, any BLE central device can connect to the bootloader and start sending a firmware update. In such a case, the bootloader will copy the firmware image to the correct location in the flash memory. However, this application will not be executed or stored any further if the update does not contain a valid signature.

A valid signature is generated by hashing the firmware image with SHA256 (256-bit secure hashing algorithm) and then signing the hash with a private key. The public key from the same key pair is flashed in the memory at the time of production. The bootloader always verifies the signature upon the completion of the data transfer from the BLE. The signature is verified by first decrypting the signature using the public key in the flash memory and then computing the SHA256 of the copied image. Next, the generated hash and the decrypted hash are compared; if the hashes match the application address is stored, and the application is executed. This method makes sure the source of the application is confirmed as well as the application is valid and untampered. Therefore, it is safe to say that updating the device firmware with a tampered application is next to impossible.

4.5 Application architecture and flow

The main application is responsible for satisfying all the requirements described in section 5.2. The application sits in the memory location described in the figure 6 and starts by initializing the system. The system initialization includes the initialization of various variables, ADC channels, power control unit and a LED. Next, the application initializes the peripherals described in the figure 5 by switching the power on to each peripheral through the PCU.

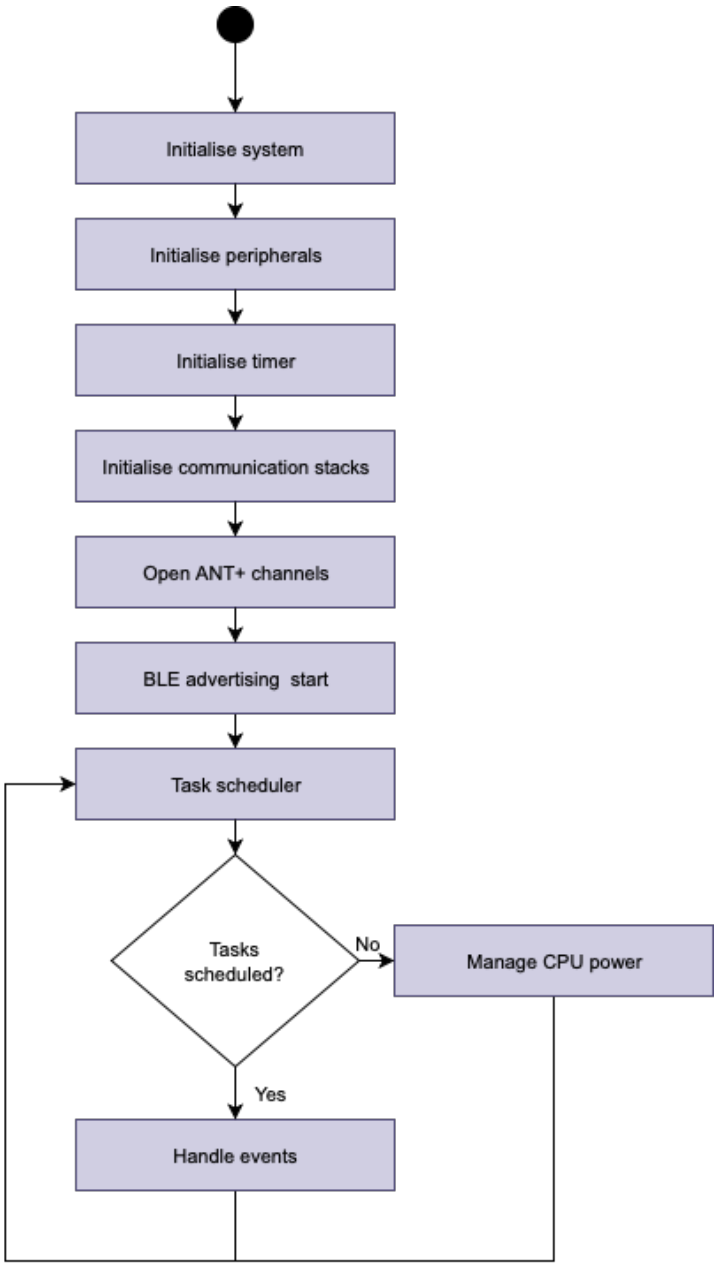

Figure 8. The flow chart of the main application

As per the figure 8, the application initializes the timer block. The block creates instances for five application timers and starts three of them right away along with a watchdog timer. At the time of the initialization, a timeout handler is allocated to each timer. The handler is called once the corresponding timer causes an interrupt. Note that all the timers run in a repeated mode, i.e. when the timeout occurs the timer automatically restarts. The description of the timers and the handlers are as follows:

- LED timer interrupts to blink the LED on-board once every second. This timer starts in the same block.
- ADC timer that is responsible for collecting information from all the ADC channels. The same ISR calculates the battery state of charge and initializes the charger if it is connected. This timer triggers every 2 seconds, and it starts in the timer initialization block.
- Data collection timer runs every 200 ms to increment a state machine. This timer also starts in the timer initialization block.
- BLE notify timer runs every one second to send a BLE notification packet to the peer device if the data is present. This timer only starts when the BLE central device establishes the connection with the device and stops when disconnected.
- Charger reset timer runs every 20 seconds to send a packet to the charger module if the external power source is connected. The charger module requires the packet to ensure safety. If the charger module on board does not receive this packet, the charging will immediately stop. This timer starts in the ADC timer handler if the charger is connected and stops in the same handler if the power source is absent.

Continuing with the flow in the figure 8, the application initializes the communication stacks for BLE and ANT+ as well as the instances for SPI, I2C, and UART. The same block initializes the implementations for TCP/IP and Message Queuing Telemetry Transport (MQTT). The implementation of the MQTT and TCP/IP is confidential and by no way the development of the author. The BLE and ANT stacks cause interrupts in the application upon the activity on the transceiver(s), and the corresponding handlers handle the interrupts.

Further, the ANT channels are opened to scan for the corresponding ANT+ profiles. The opening of channels means each channel gets a timeslot on the transceiver and scans for the appropriate profiles. In case, the transceiver picks up the data on any channel; the

stack forwards the data to the application in the form of an interrupt. Subsequently, the BLE advertising starts; where the device is made discoverable to the other device(s) scanning for the peripheral devices.

Finally, the application scheduler starts scheduling the tasks according to the priorities set. The scheduler in this application is a custom scheduler developed by authors colleagues at CompanyX and is in no way is a development of the author and therefore not described. The task scheduler deals with the connection to the server as well as publishing and subscribing of topics using the MQTT and TCP/IP protocols. The author did not handle this part of the development; therefore, it will not be described any further. The implementation of the polling of the interrupts is integrated into the scheduler by based on priorities set. As the execution of polling does not help the topic of thesis much, therefore, details are omitted.

5 Software architectures in MCU based systems

There are numerous ways of developing software for embedded applications, and there is no such thing as one architecture fits all. However, software architectures in the embedded software development as far as MCUs are concerned are still broadly classified into three types; Super-loop or polling architecture, interrupt driven architecture and multitasking architecture. It is also possible that a program might consist of a composite of these architectures.

5.1 Polling architecture

In a super-loop or polling architecture, the processor waits until the execution condition is met, executes it, and then goes back to waiting. The flow chart in the Figure 9 describes the process. Note that this is a simple setup and works only for basic tasks.

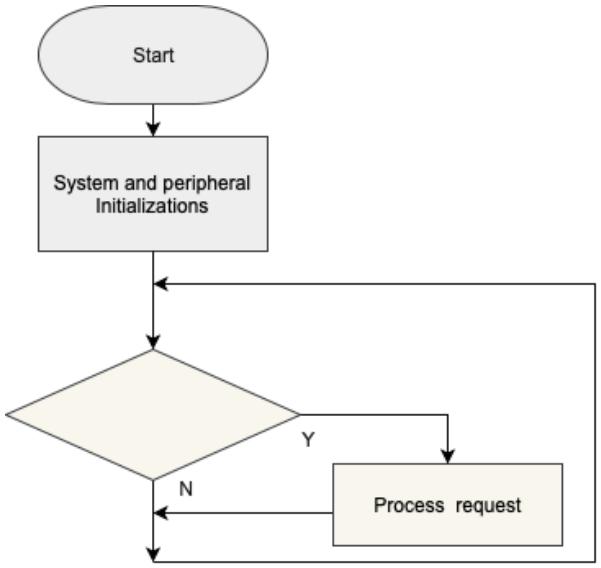

Figure 9. Flow-chart of a simple polling program [12]

In a more realistic application, a microcontroller has to handle many interfaces and support different processes. In such cases, program flow extends to support multiple processes as shown in the flow diagram in the Figure 10.

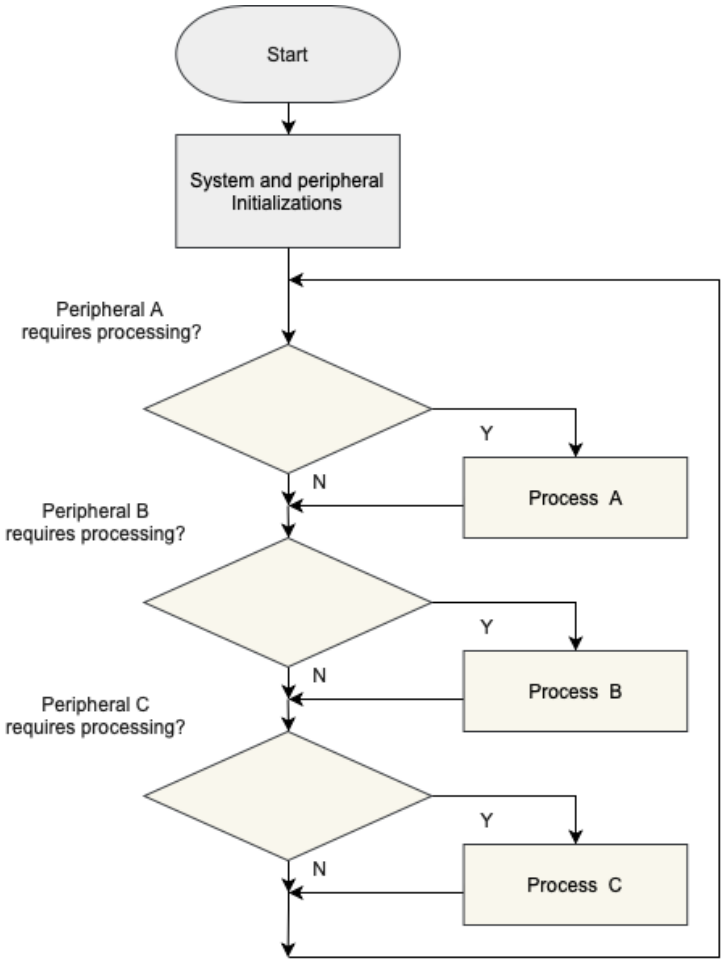

Figure 10. Flow chart of polling with multiple peripherals [12]

The polling architecture is generally used in simple applications and has several drawbacks. For instance, when the program gets complex, the polling loop architecture gets quite challenging to maintain. Also, priorities cannot be set between different peripherals which might result in reduced responsiveness. In such a case, the peripheral requesting service will need to wait while the processor is executing less critical tasks. Furthermore, the super-loop architecture is not energy efficient as the energy gets wasted when the processor is just polling.

5.2 interrupt-driven architecture

In an interrupt-driven architecture, peripherals are assigned with interrupts of various priority levels. Important peripherals are assigned a more significant priority level so that when a higher priority interrupt arrives the processor services the request immediately. The interrupt is only serviced if a lower priority service is being executed or the processor is idle. The lower priority interrupt service is halted, letting the higher priority interrupt

service to start instantly. This arrangement is more responsive than the super-loop architecture and also allows the processor to enter a low power consumption mode often called the sleep mode.

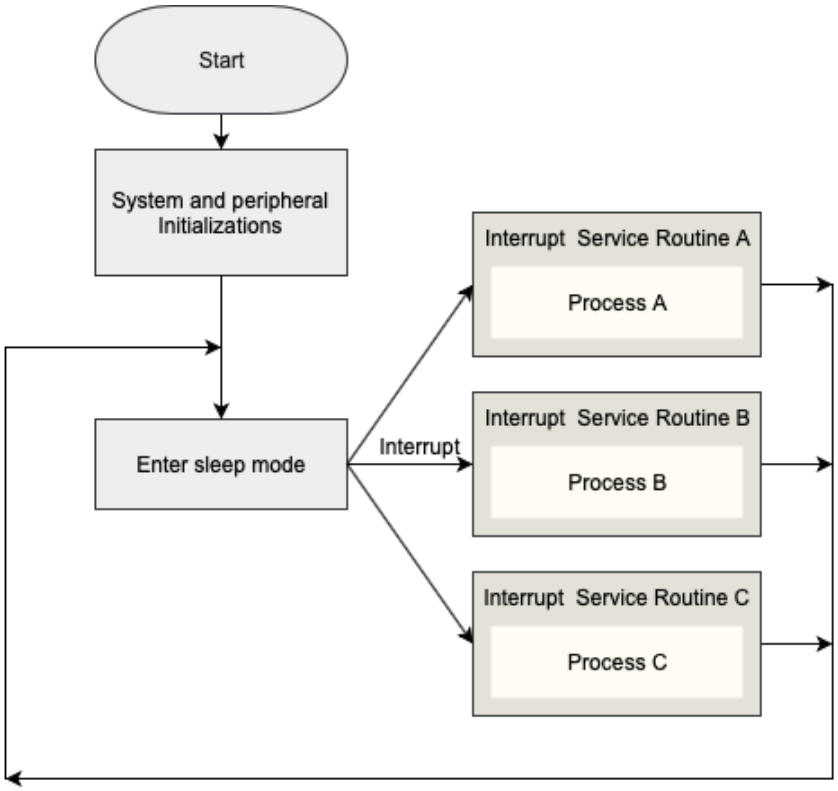

Figure 11. Interrupt-driven architecture [12]

5.3 Polling and interrupt hybrid architecture

Sometimes, data from peripheral interrupt service gets processed in two parts: the first part gets immediate attention and runs for a short time while the second part gets executed with a certain delay. Thus, a mixture of interrupt-driven and polling architecture is used to create the program. When a peripheral requests service, it triggers an interrupt service routine (ISR) as in an interrupt-driven application. Once the first part i.e., the ISR is carried out, it updates some software variables so that the second part of the service can get executed in the polling-based application code. Thus, the duration of high-priority interrupts handlers is reduced, and lower priority interrupts get a faster response.

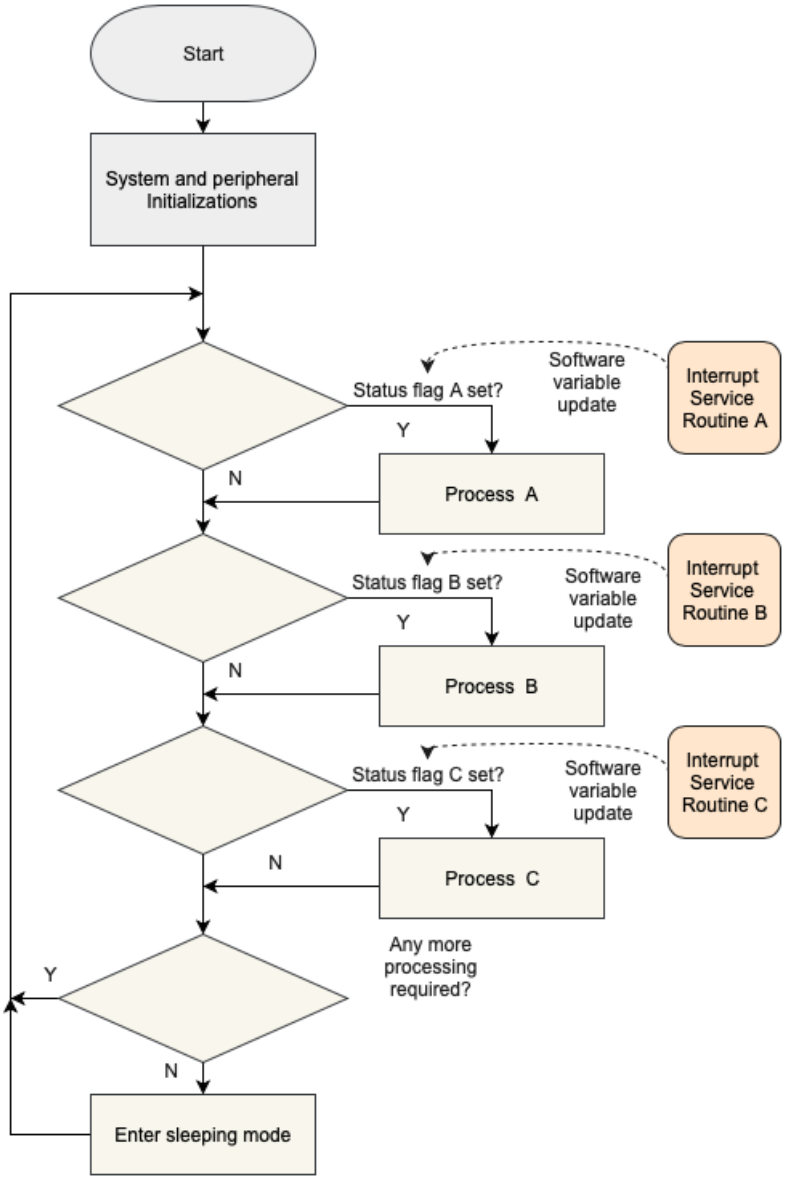

Figure 12. Interrupt-based architecture mixed with polling architecture [12]

5.4 Multi-tasking architecture

Multi-tasking architecture is generally used when the application necessitates concurrency, and a polling or interrupt-driven program structure cannot meet those demands. In such cases, the developers opt to divide the processor's time into some time slots and allocate those time slots to these tasks. While it is technically possible to create such an application by developing a simple scheduler, it is usually unreasonable to do this in practice as it is time-consuming and can make the program much harder to maintain and debug. In these applications, a RTOS can be used to handle the task scheduling.

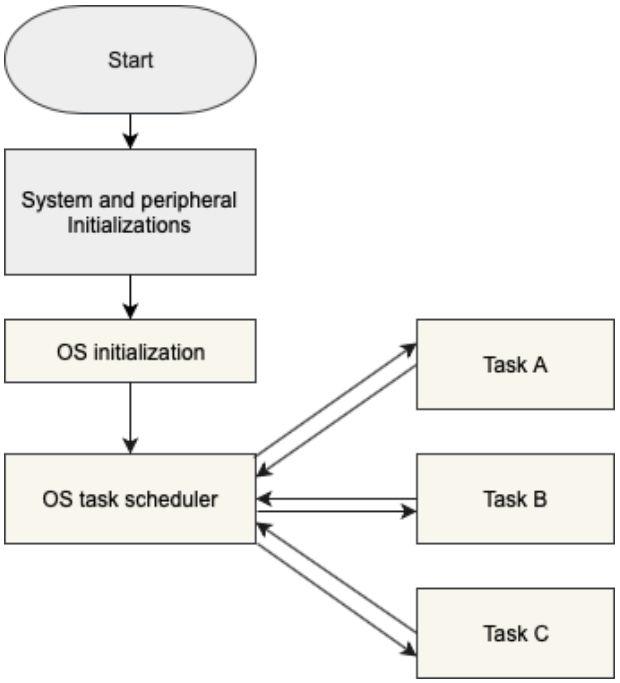

Figure 13. Multi-tasking/ RTOS based program flow [12]

An RTOS allows multiple processes to be executed concurrently, by dividing the processor's time into time slots and allocating the time slots to the processes that require services. A timer is needed to handle the timekeeping for the RTOS, and at the end of each time slot, the timer generates a timer interrupt, which triggers the task scheduler and decides if context switching should be carried out. If yes, the executing process will suspend, and the processor executes another process. Besides task scheduling, RTOSes also have many other features such as mutexes, semaphores, etc.

5.5 Architectural analysis of DeviceX

From the description of different software architectures above, it is clear that the DeviceX utilizes interrupt and polling based hybrid architecture to satisfy most of the software requirements. The bootloader strictly follows the hybrid architecture while the application uses a very basic custom task scheduler. Even though the hybrid architecture offers excellent results, developing and maintaining software with such an architecture is not pleasant especially when the application gets large. From the description of agile development in chapter 2 and the description of RTOS in the section above, it should be clear the RTOS based architecture can reduce the development efforts and promote the agile development practices more than the existing architecture.

6 Implementation of RTOS based architecture

Based on the research and analysis of this work, the author's colleagues find the arguments compelling and support the idea of testing the use of an RTOS. However, as soon as the decision of testing an RTOS was made. It became evident that some more work needs to done even to see which RTOS to test. According to ARM development portal, there are 30 [2] operating systems currently on the market each with a better sales pitch than the other. So, based on the application requirements and ease of porting to the hardware of DeviceX and online community resources the choices were narrowed down to two; purely based on online research and one visit to the Embedded World 2019 conference. Further, the choice of choosing one from the other is described in the next section.

6.1 Choosing RTOS

To choose an RTOS from the two narrowed down RTOSes, the author used a table developed by one of the presenters in the Embedded World 2019 conference. The table features the following categories for choosing an RTOS:

- The performance is a critical factor when selecting an RTOS, and several factors were analysed. Given the complexity, RTOSes require additional code and data. Therefore, memory requirements, such as flash memory and RAM are critical. Additionally, processing speed, such as interrupt latency, context switch time and power efficiency should be taken into account.
- Considering the features of an RTOS, there are many, and depending on the application some features are more important than the others. Therefore, scalability, memory protection schemes are significant factors along with interface standards conformation, i.e., CMSIS.
- The ecosystem plays a crucial role in the RTOS selection process. It ensures smooth integration, support, and product life. Also, having examples of projects and successful ports available indicates support and an active user base.
- Middleware components such as communication stacks are essential for most software developments. The more components there are easier is the development. Middleware components can indicate integration efforts as well.

- Vendor or the source of the RTOS indicates the support and the documentation. It helps to decide the lifespan of the final product as well.
- Finally, the engineering teams personal experience with the RTOS choices allows focussing on product differentiation, rather than on promotions.

The table below represents all the criteria's used for selecting an RTOS. The weights in the table represent the importance of the requirements on the scale of 1 to 5. The weighted rating total is the sum of ratings (on the scale of 1-5) given by three engineers at the CompanyX to each criterion multiplied by the weight. Then the average represents the average rating value for each criterion. It is followed by the standard deviation between the ratings of different engineers.

Table 1. Selecting an RTOS

Criteria	Weight	Zephyr RTOS			FreeRTOS		
		Weighted Rating Total	Average	Std Deviation	Weighted Rating Total	Average	Std Deviation
Performance							
Smallest RAM footprint	5	60	4,00	0,00	70	4,67	0,58
Smallest ROM footprint	5	60	4,00	0,00	60	4,00	0,00
Highest degree of determinism	0	0	0,00	0,00	0	0,00	0,00
Best meets reliability requirements	1	15	5,00	0,00	15	5,00	0,00
Minimal context switch times	2	14	2,33	0,58	12	2,00	0,00
Minimal interrupt latency	3	36	4,00	0,00	36	4,00	0,00
Lowest energy consumption	5	70	4,67	0,58	60	4,00	0,00
Features							
Best Real-time trace capabilities	1	12	4,00	0,00	12	4,00	0,00
Supports static allocation of RTOS objects	5	70	4,67	0,58	65	4,33	1,15
Most efficient memory protection	5	75	5,00	0,00	50	3,33	0,58
Easiest to scale	5	75	5,00	0,00	30	2,00	0,00
Easiest to configure features	5	70	4,67	0,58	55	3,67	0,58
Conforms to required interface standards (i.e. CMSIS)	5	75	5,00	0,00	75	5,00	0,00
Easiest to port to other MCU's and architectures	4	44	3,67	0,58	60	5,00	0,00
Most relevant safety certifications	0	0	0,00	0,00	0	0,00	0,00

Table 2. Selecting an RTOS continued

Criteria	Weight	Zephyr RTOS			FreeRTOS		
		Weighted Rating Total	Average	Std Deviation	Weighted Rating	Average	Std Deviation
EcoSystem							
Highest adoption rate in target industry	2	14	2,33	0,58	16	2,67	1,15
Most architectures supported	4	48	4,00	1,00	52	4,33	0,58
Largest and most vibrant forum community (fast to respond)	5	45	3,00	1,00	65	4,33	0,58
Fastest technical support available	3	9	3,00	0,00	15	5,00	0,00
Highest quality professional training available	2	3	1,00	0,00	15	5,00	0,00
Example projects and source available	4	60	5,00	0,00	56	4,67	0,58
Integrated development tools and plugins	4	60	5,00	0,00	60	5,00	0,00
Middleware							
File system best meets system requirements	4	60	5,00	0,00	44	3,67	0,58
TCP/IP stack best meets system requirements	5	75	5,00	0,00	75	5,00	0,00
USB stack best meets system requirements	0	0	1,67	0,58	0	2,67	0,58
Graphics stack best meets system requirements	0	0	1,67	0,58	0	2,67	0,58
Middleware requires minimal integration effort	5	75	5,00	0,00	45	3,00	0,00
Additional 3rd party tools integrated seamlessly	3	24	2,67	0,58	21	2,33	0,58
Vendor							
Best historical track record	2	10	1,67	0,58	30	5,00	0,00
Most relevant certified development process	2	30	5,00	0,00	30	5,00	0,00
Shortest support request times	4	36	3,00	0,00	24	2,00	0,00
Best strategic fit	4	32	2,67	0,58	16	1,33	0,58
Longest term support for the RTOS	4	44	3,67	0,58	48	4,00	0,00
Best code quality	3	27	3,00	0,00	21	2,33	0,58
Best code documentation	4	44	3,67	0,58	52	4,33	0,58
Engineer							
Minimal professional growth potential	2	10	1,67	0,58	10	1,67	1,15
Least amount of stress to implement	4	44	3,67	0,58	24	2,00	1,00
Personal interest	0	0	1,67	0,58	0	2,33	0,58
Minimized labour intensity	4	60	5,00	0,00	16	1,33	0,58
Least deadline constrained to get up to speed	3	33	3,67	0,58	42	4,67	0,58
Most internal resources available	3	36	4,00	0,00	45	5,00	0,00
Total	**131**	**1555**	**3,48**	**0,29**	**1422**	**3,47**	**0,33**

The results from the table are visualized on the graph below:

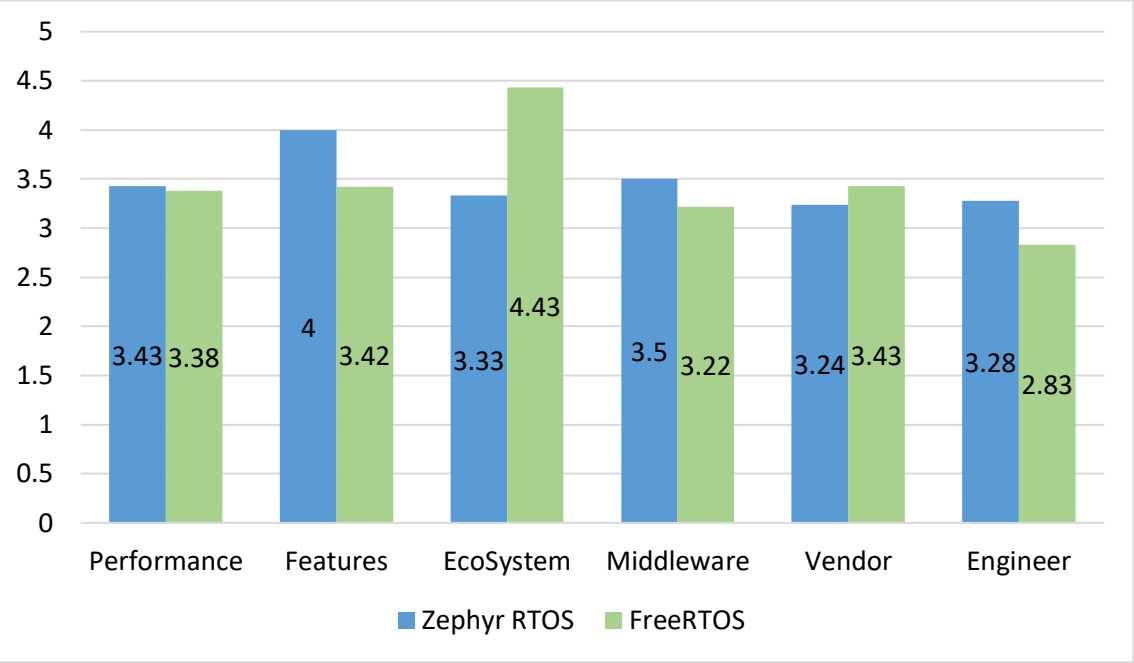

Figure 14. Graphical representation of Table 1 and 2.

With all the measures taken into account, the empirical data suggests that the Zephyr RTOS should be the choice to test and validate for further development. However, we see the both the operating systems are very close to each other and this degree of difference can easily be neglected. The author has decided to trust this method and try to integrate the Zephyr RTOS in the current hardware and investigate the complexity further.

6.2 Future plan

At the time of writing this thesis, the author had only tested some basic example projects from the Zephyr RTOS for one development board and was able to program some basic functionality on the board. Due to the time constraints for finishing this thesis as well as further development of the DeviceX the comparison of the RTOS based architecture and existing architecture was not possible. However, this work will continue after finishing this thesis, and both the architectures will be compared.

7 Summary

From the author's, experience as an embedded software developer; developing an IoT device called DeviceX led to the research of making the software development of ARM Cortex-M4 based DeviceX more agile. Compared to the sequential development processes agile development offers to accommodate the changing software requirements more efficiently, resulting in faster development times and ultimately faster time to market. Additionally, agile development preaches to develop and deploy the software in smaller chunks to increase the quality of the software. The general or mainstream software industry practices the concepts of agile development heavily, but the software requirements and standards in embedded software are very different from that of mainstream software.

The software of Cortex-M MCUs utilizes the CMSIS to deal with the incompatibility between the different software components. The CMSIS also provides interfaces for RTOSes, debug ports and system peripheral descriptions. Also, CMSIS standardizes the API calls for NVIC, SysTick, MPU, etc. So, CMSIS is supposed to be a vendor agnostic hardware abstraction layer for the Cortex-M series. However, in reality, the vendors rarely fully comply with the standards that result in incompatibility between the same architectures from different manufacturers and therefore affect software portability and reuse.

The DeviceX being the subject of interest and development of the author features multiple connectivity options including but not limited to BLE, ANT, and Cellular. The device uses a DFU capable, secure bootloader that can update the main application over the air. Hence, allowing to implement the continuous development and deployment teaching of the agile development. The bootloader software follows an interrupt and polling based hybrid architecture that is sufficient for its size and purpose. The application also utilizes the same architecture but with the addition of a simple task scheduler. However, with the growth in the functionality of the software, it becomes more and more challenging to maintain a custom scheduler and its severe limitations. It makes changing the software logic more complicated and thus makes the development less agile.

RTOS allows execution of multiple processes concurrently by utilizing a complex scheduler. The RTOSes also offer multiple middleware applications and drivers that make the use of peripherals easier and development faster. RTOSes built on top the CMSIS-RTOS can provide a hardware abstraction layer that makes porting software in the Cortex-M family easier. Given the advantages, it can be the key to making the software development for the DeviceX more agile.

References

[1] C. J. Christof Ebert, “EMBEDDED SOFTWARE: FACTS, FIGURES, AND FUTURE,” IEEE Computer Society, 2009.

[2] A. F. Joseph Yiu, “Cortex-M Processors and the Internet of Things (IoT),” ARm White Paper, 2013.

[3] P. J. M. Michael V. Woodward, “Challenges for embedded software development,” in *IEEE International Midwest Symposium on Circuits and Systems*, 2007.

[4] M. Rouse, “WhatIs.com,” TechTarget, January 2015. [Online]. Available: https://whatis.techtarget.com/definition/ARM-processor. [Accessed 27 March 2019].

[5] J. Yiu, “ARM® Cortex®-M for Beginners,” ARM White Paper, 2017.

[6] J. Yiu, “ARMv8-M Architecture Technical Overview,” ARM white paper, 2015.

[7] E.Moorits, “Embedded Software Solutions for Development of Marine Navigation Light Systems,” Tallinn University of Technology, Tallinn, 2016.

[8] D. M. S.Balaji, “WATEERFALLVs V-MODEL Vs AGILE: A COMPARATIVE STUDY ON SDLC,” *International Journal of Information Technology and Business Management,* vol. 2, no. 1, pp. 26-30, 2012.

[9] T. Robal, “IAF0320 Computer Systems Engineering LECTURE 5,” Tallinn, 2018.

[10] cPrime, “Scrum FAQ,” cPrime Inc, [Online]. Available: https://www.cprime.com/resource/white-papers/scrum-faq-2/. [Accessed 2 May 2019].

[11] T. Punkka, *Agile Methods and Firmware Development,* Helsinki: Helsinki University of Technology, Software Business and Engineering Institute, 2005.

[12] J. Yiu, in *The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors* , Cambridge, Newnes, 2014, pp. 42-62.