OPEN653

- An OS Implementation based on the ARINC 653 Standard -

Project Report

Group d707e16

Aalborg University
Department of Computer Science
Embedded Software Systems

Department of Computer Science
Aalborg University
http://www.cs.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
An OS Implementation based on the ARINC 653 Standard

Theme:
Operating Systems

Project Period:
Fall Semester 2016

Project Group:
d707e16

Participants:
Anders Normann Poulsen
Charles Robert McCall
Gabriel Vasluianu
Jakob Østergaard Jensen
José M. R. S. d'Assis Cordeiro

Supervisor:
Ulrik Mathias Nyman

Copies: 1

Page Numbers: 106

Date of Completion:
21st December 2016

Abstract:

ARINC 653 addresses the increasing number of microcontroller units in aviation systems. By unifying applications under a single API, and using time and space partitioning, it is possible to use a single computing device to support multiple safety critical applications. This project describes the implementation of the ARINC 653 standard, as an Operating System, on an embedded platform, Mini-M4 for STM32.

Preface

This report is written by the d707e16 group following the Embedded Systems Software master programme at Aalborg University. Its subject is the 7th semester project - An OS Implementation based on the ARINC 653 Standard. This resulted in developing a system that fulfils the study regulations.

The group would like to thank Ulrik Nyman for great guidance.
The group is also grateful to AAURacing, at Aalborg University, for letting us use their build configuration and basic drivers for the hardware used in this project.

Aalborg University, 21st December 2016

Anders Normann Poulsen
<apouls16@student.aau.dk>

Charles Robert McCall
<cmccal16@student.aau.dk>

Gabriel Vasluianu
<gvaslu16@student.aau.dk>

Jakob Østergaard Jensen
<jaje11@student.aau.dk>

José Maria Reis Simões d'Assis Cordeiro
<jreiss16@student.aau.dk>

Reading Guide

In this report, references are written using the Harvard method [last name(author or company),year(if available)]. Sources are listed alphabetically in the bibliography. Figures, Tables, Equations, and Listings (code snippets) are numbered after the chapter they are in, for example, the first figure in chapter two is called 2.1, the next 2.2 etc. All illustrations and figures are made by the group members unless otherwise stated. All data sheets and code can be found on the zip archive electronically uploaded with the report.
A prerequisite for reading this report is basic knowledge in the computer science field.

Glossary

ABI	Application binary interface
APEX	Application/Executive
CCM data RAM	Core coupled memory data RAM
COM	Communication port
COTS	Commercial off the shelf
CPU	Central Processing Unit
CMSIS	Cortex Microcontroller Software Interface Standard
EWI	Early Wakeup Interrupt
FPU	Floating-point unit
GB	Gigabyte
GPIO	General-purpose input/output
HAL	Hardware Abstraction Layer
IDE	Integrated development environment
IMA	Integrated Modular Avionics
IoT	Internet of Things
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group
KB	Kilobyte
MB	Megabyte
MCU	Microcontroller unit
MMU	Memory Management Unit
MPU	Memory Protection Unit
NVIC	Nested Vectored Interrupt Controller
OS	Operating System
PLL	Phase-locked loop
RISC	Reduced instruction set computing
RTC	Real-time clock
SRAM	Static random-access memory

UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus

Contents

CHAPTER 1

Introduction

Embedded devices are becoming more powerful and less expensive year by year. The majority of microprocessors produced end up as components of embedded devices, ranging from mobile phones to traffic lights, from digital watches to medical apparatus. These devices are tailored for executing specific tasks, thus requiring the developers' skill and commitment to create reliable systems. The diverse applications of these systems divide them into different categories, in regards to how dependable they have to be. A wristwatch skipping a second has no severe consequences, while the failure of a surgical robot has to be avoided at all costs. Technical standardization addresses these requirements, to better reason about the expected reliability of such systems; some standards such as ARINC 653[19] enforce high reliability while others focus on other metrics.

One might argue that these regulations are loosely formulated and too limited in scope. For example, if one compares the existing standards for the avionics and automotive fields, there are large discrepancies. The standards in use in the automotive industry are applying certain rules from the avionics regulations, but only to a certain extent[13]. These two fields seem to be converging in terms of safety requirements as the automotive industry develops. This gives rise to the idea of applying these avionics safety critical systems to other fields. There is potential for reducing weight of products with a single computing device[50], facilitating more safety procedures or increasing portability of applications across generations of products.

This project's cornerstone is developing an Operating System based on a defined subset of the ARINC 653 standard, that facilitates running safety critical applications compliant with the ARINC 653 standard.

The two main areas of interest are:

- The standard itself, that specifies an IMA (Integrated Modular Avionics) approach to run applications
- The physical platform that would contain the system and serve as a research launchpad

The Analysis chapter provides more details about the software specification that need to be fulfilled, as well as the different available hardware options.

CHAPTER 2

Analysis

This chapter includes information about the standard, the hardware going to be used as well as the programming The ARINC 653 requirements are analysed, while looking at existing projects and other relevant standards in the field of avionics. Based on this, decisions can be made regarding the following stages of Open653 's development, and narrowing the project's scope.

2.1 Real-time systems

A real-time system is handling data as it comes in. Besides the usual computations needed to be done on data, the worst case execution time of the computation is also important. This means that the applications are scheduled in a manner that would allow them to respond predictably. Real-time systems are developed in order to answer requirements of different fields of work where time is critical: military, automotive or avionics.

2.2 Integrated Modular Avionics

In the avionic industry, the devices and subsystems used for controlling the peripherals have to be reliable and operate using fixed time constraints. Integrated Modular Avionics is a term used to describe a distributed real-time computer network, deployed in the avionics industry[10]. Modularity, keeping track of the time and criticality levels are the main characteristics of such a network. IMA also aims to centralise data processing by routing sensor data to only a few compute nodes, rather than processing data at the sensor. This implies that the applications running would share the system's resources following a set of well defined rules.

The main benefits of using the IMA concept are: aircraft weight reduction[50] and lower maintenance costs[53]. There are certain guidelines how such a system should be designed and implemented, as well as standards that specify how space and time should be partitioned.

2.3 ARINC 653

ARINC 653 is a software specification that allows a system to host multiple avionics applications on the same hardware, while being able to execute them independently. This is achieved by partitioning the system[20].

2.3.1 Overview

ARINC 653 is said to *bring a new quality in real-time systems development*[53]. This is achieved through its API (APEX), the interface between the OS and the software applications. Applications are residing in partitions, which have their own memory and scheduled processing time slots. These partitions can not interfere with each other and can not take resources from each other. If an error in a partition occurs, it will only crash itself; the remaining partitions will not be affected.
The following is a brief description of what the standard specifies.

Modules refer to the aviation concept of IMA, where a module can be seen as a piece of hardware, controlling other subsystems. A module can contain one or more processors. However, this does not fit well with the concept of partitions, which have to run on a single processor[21]. A processor used for ARINC 653 applications should have sufficient processing power and access to the required inputs and outputs, as well as memory and time resources. Besides this, the processor should be able to isolate a partition from the others, if it fails[22].

Partitions are program units designed to obey space and time limitations set by the developer. The way partitions are being run by the processor, is in a static, cyclic manner meaning that they do not have any priorities. One partition should only have writing rights to its own memory space[23]. The resources used by the partitions are specified at build time[24]. The standard differentiates between partitions that run applications and system partitions. The latter are optional, and can be used to contain services not provided in the APEX[53].

Processes are the parts of the program running on partitions. One partition may contain one or more processes. These may operate concurrently, and may operate in a periodic or aperiodic manner. A process's characteristics are: data and stack areas, program counter, stack pointer, priority and deadline. This makes

them resemble threads in the POSIX[68] process model. Certain attributes are statically defined, and cannot be changed during operation[25]. Process management is done with the use of APEX calls. Processes can be in different states: dormant, waiting, running, ready. Their scheduling is done in such a way that the process in the ready state with the highest priority, is always executing when the partition is active. This entails processes being preempted at any time, by the process with a higher current priority[26].

Time management is very important in real-time systems. Time keeping is needed for partition and process scheduling, for delays, deadlines as well as for communication time-outs. Partitions receive time-slots during system configuration, and processes receive time capacity[1] when they are started. The difference between the two brings the need for having extra APEX calls to control and monitor how processes are behaving[27].

Memory spaces are defined during the system configuration. These can not be changed later on.

Interpartition communication is an important aspect of ARINC 653. It provides a framework for processes across different partitions to communicate with each other. This applies to partitions placed on the same module, on separate core modules, as well as communicating with non-ARINC 653 modules. This communication is based on message passing. Channels are used as logical links between partitions. Partitions can access channels through ports, which allow monodirectional access to a channel[28]. These ports are defined during system configuration and are tied to both a certain channel and a certain partition.

Intrapartition communication allows processes inside a partition to communicate to each other. This way the communication is done more efficiently than having to use a global message passing system. This can be done by implementing buffers and blackboards, or semaphores and events[29].

Health Monitor Has the purpose of monitoring, reporting and isolating hardware or software faults and failures. Its responsibilities are divided into the following levels: process, partition and module. The health monitor has error response mechanisms to take action in case typical error occurs at any of these levels [30].

The standard allows portability and reusability of applications. However, these applications first have to be integrated in the system. This is done by giving the

[1]Time given to them to satisfy their process requirements. This is an absolute duration, not execution time

partitions the access and resources needed, in the **system configuration** phase, in order to satisfy the applications' requirements[31].

2.3.2 Decision

The idea is to bring such an operating system based on ARINC 653 in to the wider embedded world. The focus will now change to understanding if, and how other projects have approached this idea.

2.4 Other existing standards and ARINC implementations

While ARINC 653 defines the architecture of a system for safety-critical avionics applications, there are other ways to categorize the reliability of such a system. Usually this is measured by looking at the number of failures occurring in time frame. The following standard attempts to grade a systems reliability.

2.4.1 DO-178B

DO-178B[66] is a set of guidelines to be used for software assurance in safety-critical avionics systems. Use of the standard can determine the reliability of the software in an airborne environment. There is a *safety assessment process* and *hazard analysis* to determine the result of failures in the system. Applications are graded with a *software level* of A to E, where A is the best level to be achieved and E is the worst. A level A compliance means the system has been verified and validated against catastrophic failures, whereas an E compliance has been verified against trivial failures. Its successor, DO-178C[67] was released in January 2012.

2.4.1.1 Certification of ARINC 653

The concept of having partitions to host avionics applications, is already seen as a step forward in terms of safety. Applying the DO-178B standard on an ARINC implementation would certify that this is ready to be deployed in the avionics field with a grade. In fact this is what similar projects are doing.

2.4.2 Alternative ARINC 653 platforms

There are a number of ARINC 653 implementations available, which is not surprising, having understood the benefits of IMA. These are divided open source OSs and COTS[2] OSs. An existing platform could be used as a starting point for the implementation of Open653.

[2]Commercial of the Shelf

ARLX[43] is an open-source platform available. It is a bare-metal hypervisor for ARINC 653 based on the Xen hypervisor, with a Linux partition as the main virtual machine (in Xen terminoligy, domain 0). It has been developed in accordance with DO-178C.
Commercial platforms such as VxWorks 653 (DO-178C certified)[52], PikeOS (DO-178 compliant)[2], LynxOS-178 (DO-178B certified)[44] support ARINC 653.

As an example, **VxWorks 653** is marketed as *The Industry's Leading Operating System for Embedded devices*[52]. It contains an ARINC 653 compliant platform and is compliant with DO-178C to level A.
As mentioned, there is no surprise that the existing ARINC 653 implementations are all DO-178 certified. While it is important for them to work properly, the compliance to ARINC 653 and the certification of a high level in DO-178 shows that they are safe to use in critical applications.

2.4.3 Decision

The research on the existing ARINC 653 platforms concluded with at least four options. The commercial OSs are excluded from the start since they are not in the project's scope. The only remaining variant, the ARLX has to be excluded as well, since it is a hypervisor, and does not fit the scope of the Masters programme. There has been gained in-depth knowledge about certification and other standards, but for the rest of this document, only the ARINC 653 specification and design consideration will be used. The fact that the group does not work on top of an existing implementation increases the degree of freedom for choosing a hardware platform to fit the scope of the project and of the education programme.

2.5 Hardware

Having the ARINC 653 specification basic prerequisites in mind, the different options for hardware are investigated in this section.

2.5.1 The rise of embedded devices

In the world of embedded computing, the computing platforms vary a lot, compared to the world of workstations and desktop computers. Traditionally, embedded computers are used for small specialised static tasks and not required to run an operating system with multiple processes. For this reason many embedded systems are arranged as a SoC, with limited amounts of ram, flash and a low CPU clock, not capable of supporting large computer screens and high speed networking.

Improved technology however, have brought powerful new CPUs of architectures used in the embedded world such as MIPS and ARM. The latter now being used in increasingly many levels of computing, from low power single purpose embedded systems, to workstations and supercomputer installations with graphics and high speed networking requirements. For more powerful computers this might represent both a shift from the widely used CISC architecture to the smaller RISC architecture and a new focus on power efficiency.

For embedded devices this change has meant more capable systems and an increasingly narrowing gap between embedded and workstation computing[11][8]. Capable low power devices and cheap manufacturing of this new generation of RISC based computing systems, has made it possible to build computers to solve problems on all scales using the same, or similar, platforms. This has brought a surge of development for the middle layer of computing, a little more capable than traditional embedded devices and a little less capable than the typical workstation.

Developing a system to operate in an embedded environment with real time requirements and low overhead peripheral interaction, yet with the flexibility and multitasking capabilities of a modern operating system, puts some lower limits on the capabilities of the hardware.

2.5.2 The lower bound on systems

Overall these requirements must be met by the hardware:

- Be fast enough to run multiple applications and the operating system
- Have sufficient main memory to support the stacks for all running applications and the operating system
- Must come with either an MMU or MPU to provide protected regions in memory for separate contexts

The first two requirements depend almost entirely on the tasks being solved. The more applications being run and the more resource demanding they are, the more CPU time and memory they are going to utilize. The requirement to provide an MMU or MPU to safely segregate blocks of memory between applications and the operating system, puts a lower bound on the type of hardware required.

More capable systems may provide an MPU for basic memory protection. The smallest embedded systems typically do not provide any memory protection. Only being required to run single purpose applications where all drivers and interfaces

are compiled and executed as a single binary, hence they can expect no foreign element disrupting their execution. More powerful systems may come with a MMU, providing a memory sandbox of each application.

2.5.3 Some of the options out there

The best way to decide on a hardware platform to work on is to explore some of the options out there. Implementing an operating system from scratch, the hardware should not be in the complex to work with.

In appendix A is a brief list of hardware platforms listed in no particular order. Every unit has been selected on the premise of possessing either an MMU or MPU for the space partitioning feature in the OS, also the candidates vary to some degree in capability and complexity.

These are the evaluated hardware platforms:

Creator Ci40 is a platform meant for IoT development. It comes with easily available ports and connections for embedded development and networking over protocols like Ethernet and Bluetooth. It is built around the cXT200 chip which is based on a JZ4780 550MHz dual-core MIPS CPU[64]. It comes with an MMU, is capable of running Linux and is generally more powerful than many low power embedded platforms.

MINI-M4 for STM32 is an ARM Cortex-M4 based development board containing a STM32F415RG microcontroller. It provides a simple pin layout and fits well into a breadboard for connecting external devices and comes with a single USB port to deliver power and to be programmed through, with an installed bootloader. The STM32 chip comes with a MPU and a single core CPU capable of running 168 MHz, but limited to 120 MHz using the on-board bootloader[47]. Available is a downloadable hardware abstraction layer (HAL) library from chip manufacturers website, which comes with a number of implementation examples and documentation[60].

MINI-M4 for Tiva is similar to the MINI-M4 for STM32 above, based on the same ARM Cortex-M4 architecture, but using the TM4C123GH6PM chip from Texas Instruments instead. It has a pin layout similar to the MINI-M4 for STM32, a USB connection for power and programming and comes with an MPU, but only runs at 80 MHz[48].

Raspberry Pi 3 is based on a 1.2GHz 64-bit quad-core ARMv8 CPU which makes it available for both embedded use and common desktop computing tasks[11]. The

Raspberry Pi lacks any comprehensive documentation on its register layout, and being a higher end device compared to MINI-M4 and Creator Ci40, it is presumably more complex to work with.

2.5.4 Settling on a platform

Obviously a list longer than the one seen above can be compiled. Although the hardware used is not the primary concern to the project and as such the goal is to show examples of known platforms, available for purchase in Denmark, that all hit within the range of what can be considered embedded devices.

The two MINI-M4 devices represent a lower, but capable end of the performance spectrum, while the Raspberry Pi and the Creator Ci40 are more capable. All four devices have the necessary features to run a simple low power OS, but building the OS gives rise practical concerns to the complexity and ease of setup.

Focusing on core OS features, only a few basic peripherals like timers, MPU/MMU, UART and basic IO are necessary for the project. The ARINC 653 does not provide a solution to scaling the operating system or containers across multiple cores, hence more capable hardware is likely to see a low utilisation. Since the project is focused on the implementation of an OS and less so the partitions running on it.

The MINI-M4 modules are more attractive choices from the utilisation standpoint. A comprehensive datasheet is available for both of them, making it easier to program them from scratch. Some experience with the STM32 is present within the project group on how to setup a Linux based tool-chain. The manufacturer of the STM32 provides sample code implementations as well as hardware abstraction layer libraries, which would give a head start on the project.

2.5.5 Decision

This leads to the decision of using Mini-M4 for STM32 as the hardware for Open653.

2.5.6 Communication with the board

In order to get started with the development of the OS, there is a need for communication with the board. Due to previous experience with this type of hardware, the following components have been ordered:

- ST-LINK/V2 in-circuit debugger/programmer for STM8 and STM32 [59] - to give access to the JTAG port
- FTDI TTL-232R-3V3 cable [42] - to create a communication port

2.5.7 Helper libraries

When starting to write software, one needs to decide how to make this process easy and transparent. This is very dependent of the project's requirements such as time frame, performance, security and reusability. One possibility would be to work on the "bare metal", and use the chip's reference guide[56], and maybe part of the ARM architecture reference manual[6].
Another way could be to use some libraries that define the I/O and work on a higher level. ARM provides libraries for their architecture as well as the vendors providing libraries and examples on how to use their hardware.

2.5.8 The legacy from the AAU Racing project

This project has some roots in another project that has been going on for some years at Aalborg University [1]. The choice of hardware was slightly influenced by this, as well as the tools and libraries used throughout the project. The big advantage of this, is that by having the experience and knowledge the project could get a head start. Before any hardware was ordered specifically for this project, the AAU Racing team lend the group a platform to work on.

2.6 Programming language choice

The languages Ada, C and C++ are taken in consideration as an embedded language.

Ada was created during the 70's by the Defense Department of the USA. It was created to reduce the number of languages used between departments. It is a high level language with real-time capabilities.

C is a simple low level language especially popular in embedded systems. Like Ada it was created in the 70's and among the most popular languages used today. C is easy to use in projects favoring memory and instruction control, as is favored with ARINC 653, where resources are allocated statically.

C++ is a multi paradigm language described by Bjarne Stroustrup as “C with classes”[63]. Created in the late 70's and first standardised in 98. It is a language with both high-level object oriented features as well as low-level memory management. It is mostly compatible with C. C++ contains many extra features compared to C, such as classes. These features can both benefit and complicate code.

For the passing of the schema, a higher level language can be used since it is running on the computer used in development, hence there is little concern for op-

timisation and memory usage. The schema will be translated into a usable format to the embedded system. Once translated it will be compiled and uploaded to the embedded system.

Python is an easy high level language to use. There is very little boiler plate code involved, it is not typeset, nor does it need to compile before it is executed. It has many libraries to interact with structured data types such as XML and JSON, which makes it a well suited option for developing the schema parser.

2.6.1 Decision

Python is chosen as the language to parse the schema because of its simplicity and because it is cross-platform. C is chosen for embedded development because of its simplicity and the level of experience within the group, with the language.
The choice of these languages have the added benefit that the resulting code can be compiled from all three major PC operating systems.

CHAPTER 3

Problem statement

The goal of this project is to develop an ARINC 653 operating system called Open653, in order to facilitate time and space partitioning of applications in different industries, based on the concepts defined in the Introduction.

The goal gives rise to the following problem statement:

Can an OS implementation based on a subset of the ARINC 653 standard be developed on a Mini-M4 for STM32?

Based on the knowledge acquired in the Analysis and limited by the time scheduled for the project, this problem can be expanded into the following sub-questions:

- Which features must be developed to make Open653 functional and testable?
- How can this system be developed without the use of pre-existing implementations?
- How is this system implemented using C and Python?
- What are the main obstacles when developing such a system?

CHAPTER 4

System Design

In this chapter the project is further analysed. Diagrams are given and other notions are described, setting up the premise for implementing Open653 based on the ARINC 653 standard.

4.1 System Overview

As a means to both explain the system and as a practical development model, the system can be thought of as being made up of layers. Each layer being an abstraction of functions in the layers beneath it, with the partitions at the top and the hardware at the bottom. The model in figure 4.1 is defined by the ARINC 653 standard and expanded upon here with drivers being built on top of the HAL library, instead of directly on top of the hardware, creating an extra layer of abstraction.

Partitions are independent program units that contain applications. These applications can in turn interact with other systems. The partitions are achieving this while separated from one another.

APEX defines the methods by which the partitions talk to other modules and interact with other partitions or the OS, running on the same core module or connected to other ARINC systems.

OS kernel provides the infrastructure for partitions to operate. This includes their scheduling, message passing and processes the APEX calls.

Schema contains the data necessary for the system initialisation.

Drivers define the way the hardware runs. They serve the OS with methods to create an environment in which partitions can run and scheduled without influencing one another.

HAL and CMSIS are libraries that provide hardware abstraction layers for setting up the hardware.

Hardware the physical platform on which the OS is executed. This includes the CPU and its peripherals.

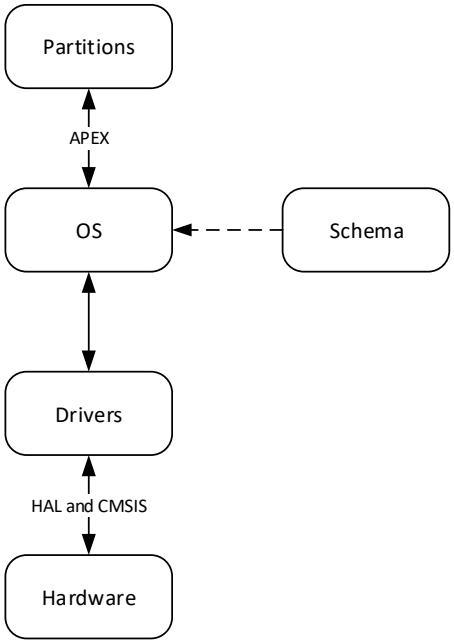


Figure 4.1: A simple system overview

4.1.1 Essential components

The rest of the chapter describes the layers of the system from the bottom up. The hardware specifications are outlined, together with some essential features needed to make Open653 functional and testable as required in chapter Problem statement. The overall problem being the implementation of an ARINC 653 OS, application execution and control is the end goal. To facilitate this, the following components must exist:

- Hardware platform

- Interfaces to interact with the hardware
- A program to parse system configurations from a schema to the kernel at compile time
- A scheduler for partitions
- A scheduler for processes
- Partition communication
- System calls to handle APEX calls

The design of these components is described in the rest of the chapter. Memory separation is not an essential feature for facilitating the execution of applications, but is essential to separate them for security and reliability reasons, thus memory protection is also described.
The features "Health Monitor" and "Intrapartition Communication" have been excluded from the project to limit the scope to a manageable amount.

4.2 Hardware

As mentioned in the Settling on a platform section, the hardware used for the development of this system is the Mini-M4 board. It is build around the STM32F415RG microcontroller[61], based on an ARM Cortex-M4 core. This is a 32-bit RISC architecture CPU using the ARMv7M instruction set[6]. Additional information was found in the ARM technical reference[4]
The chip provides two interfaces for debugging:

- JTAG Debug Port (JTAG-DP)
- Serial Wire Debug Port (SW-DP)

There was no thought process involved in choosing one of the two. The JTAG interface is enabled by default, and worked right from the early phases of the project. In some situations, SW would have been preferred to JTAG, due to physical space constraints. JTAG uses 5 wires, while SW only uses 2. Also, the IDE of choice can have some limitations on the debugging platform.
JTAG is actually an association[1] that develops standards about how physical circuit boards should be tested. However, for the rest of the document, JTAG will be referred to as the main communication interface for the Mini-M4 board. This will mainly be used for flashing the program on the chip and debugging it. In order for

[1] Joint Test Action Group[41]

a computer to access this interface, the STLINK-V2 in-circuit debugger/programmer[59] is used, through USB.

Another way of communicating with the chip is the serial communication. For this purpose the UART will be used. This is a common peripheral for embedded devices, but it needs to be set up before it can be used. This involves initialising the UART, configuring the data format and the transmission speed and sending the actual data. In order to see this data on a computer, one could use a serial-to-USB adapter [42]. This creates a virtual COM (Communication port), that is accessed using a serial console such as PuTTY[2].

The final setup of the Mini-M4 board comprises of the 5 wires from the JTAG, the 2 wires from the UART, the mini-USB cable that provides power and a breadboard to accommodate electrical connections.

4.3 Memory Map

The STM32F415RG microcontroller has 1 MB of flash memory. The flash is non-volatile memory, used for storing programs and data.

There are 192 KB of RAM, divided into 128 KB of SRAM and 64 KB of CCM data RAM. The SRAM is volatile memory, where the *static* tag means that it does not need to be refreshed in order to keep its state. This is divided furthermore into a section of 112 KB, and another of 16 KB, which can be useful if one needs to boot from RAM[49]. They are adjacent in address space and can be treated as one block. The memory has a feature called bit-banding, in order to let the system perform atomic operations on bits[3].

The other 64 KB of CCM data RAM are always ready for access by the CPU, but cannot be accessed by the peripherals through direct memory access. Beside the 192 KB of SRAM, there are 4 KB of SRAM used for backup purposes. These won't be used, since they are not within the scope of the project.

Figure 4.2 presents a simplified model of the memory map implemented in the STM32F415RG microcontroller[62]. The blocks on the left column are legacy of the ARM Architecture. This provides 4 GB of addressable memory. Some of these regions are correlated to the microcontroller's actual memory, as it can be seen in the column on the right. Each block in this column contains their start and end addresses, on their side.

[2]Free and open-source terminal emulator, serial console and network file transfer application

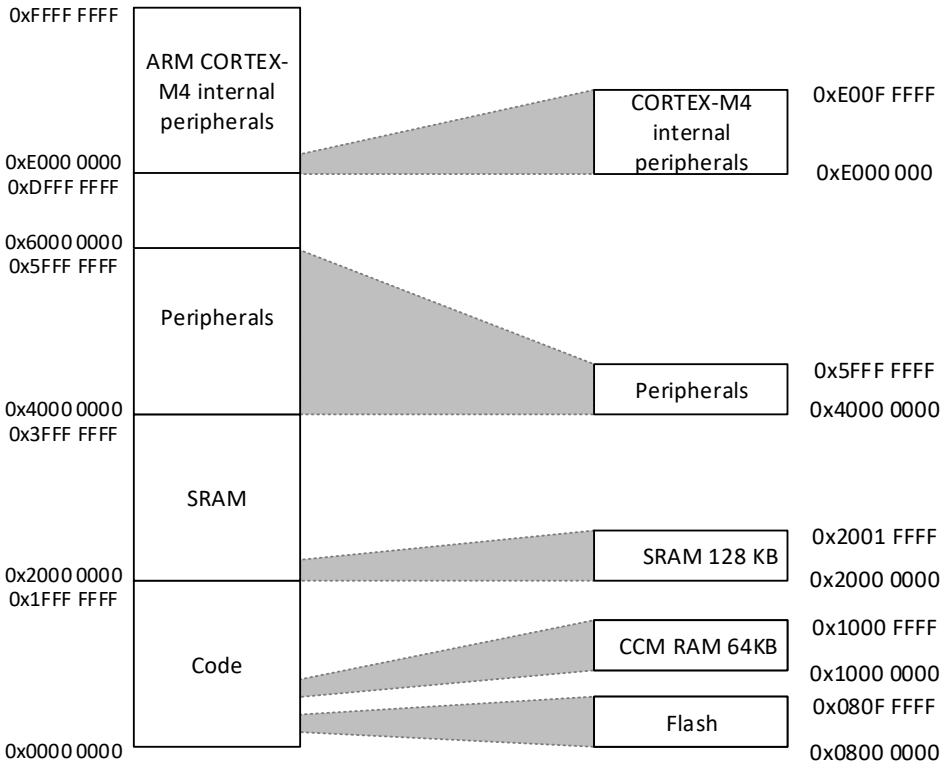


Figure 4.2: Memory map

The ARM relies on memory mapped IO to communicate with peripherals, as seen in the upper two blocks of the memory map.

4.4 HAL and CMSIS

These two libraries are the workhorses when dealing with the board's peripherals. By using them, the development process usually has a quick start, removing the steepness of the learning curve.
HAL (Hardware Abstraction Layer) is provided by the chip manufacturer - STMicroelectronics. The source files are available at their website, along with its documentation [60]. The CMSIS (Cortex Microcontroller Software Interface Standard) can be seen more as a framework, than a library [7]. It is provided by ARM, the company developing the chip architecture.

As seen in figure 4.3, CMSIS defines the data structures and address mapping

for some of the ARM peripherals. Besides this, it can be used to configure the microcontroller oscillators, as well as providing support for the instrumentation trace during debug sessions [45]. HAL is build on top of CMSIS, which defines exceptions specific to the chip. HAL uses these definitions to configure interrupts (e.g. SysTick timer).

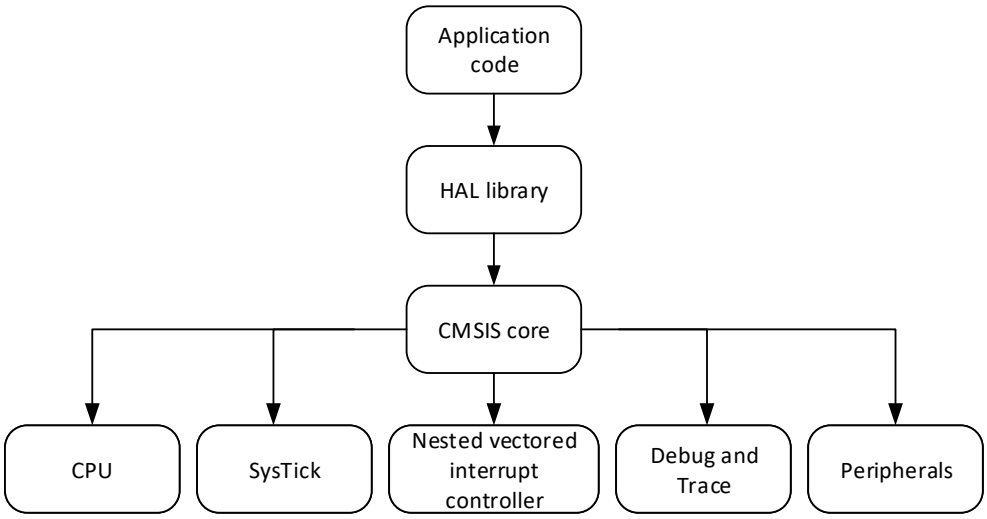


Figure 4.3: The structure of the libraries (inspired by: [54])

4.5 Drivers

The drivers controlling the hardware would have to be implemented at an early stage of the project, in order to have access to the low level subsystems. The following list contains the drivers that would provide the OS a basic interface to access the hardware.

System Clock is used as the CPU frequency, and is scaled down to provide a clock source for peripherals.
The system clock relies on an oscillator. There is a multitude of choices, ranging from internal or external oscillators, to high or low speed crystals. In some cases, certain peripherals can use their own, independent clock sources that also have to be configured

UART used by the system to communicate with the outside world. This can either receive or transmit data to another UART devices

LEDs mainly used for debugging. They are connected to two of the GPIO pins

Delay is a simple function, based on calling another function from the HAL library. It delays the execution of the program by a number of milliseconds. Mainly used for debugging

MPU provides the system a way to manage its memory. This is required by the ARINC 653 standard. Operating systems built on more advanced processors generally use an MMU

Timing has the purpose of keeping track of relative time. As the standard specifies, it should be counting nanoseconds

Watchdog timer necessary to ensure recovery in case of a hardware fault or program error. If this happens, the program would be restarted

RTC has the purpose of keeping track of time accurately. Usually it uses a 32,768kHz oscillator as a clock source, which is the standard in the field

These drivers should be implemented as standalone C files. They could then be used by including their header files where needed and calling their functions. The basic functionalities of a driver are:

- initialisation of the peripheral
- usage of its resources
- de-initialisation (if needed)

4.6 OS kernel

The OS kernel has the purpose to link the APEX interface with the hardware. This is a monolithic kernel. The following sections contain the kernel's features and functionalities.

4.6.1 Scheduler

Schedulers are used to allow a system to run multiple processes concurrently on a single CPU core. A scheduler decides which process to execute by an available processor, according to predefined rules. Based on these rules, two or more processes may run concurrently, sharing CPU time. The ARINC 653 specification defines two levels of scheduling, a partitions level scheduler and a process level scheduler. The algorithms for both are strictly defined.

The partition level scheduler uses manual scheduling. Time is split into fixed length intervals, major frames. Each partition is given one or more windows (i.e. time slices) in the major frame. Each window has a start time relative to the time of the major frame and a duration. Windows are set up by the system integrator in the XML file[32]. When a partition has run for the duration specified, the scheduler switches to the partition containing the next window. The switch will happen regardless of what the partition is doing at that time; a partition can not overrun its allotted time and thereby cause other partitions to overrun their deadlines.

The process level scheduler is a pre-emptive priority scheduler. All processes will always have a priority. Processes can only be scheduled when the partition they belong to is scheduled by the partition level scheduler. The process scheduler selects the process with the highest priority that is also in the ready state. If two or more processes have the same priority, the one that became ready first is scheduled[33].

The ARMv7M architecture has two special interrupts that are especially useful for scheduling: SysTick interrupt and PendSV interrupt. The SysTick interrupt is triggered at a regular interval, and can be used for time based scheduling. The PendSV interrupt is triggered by software and can be used to trigger scheduling from other interrupt service routines, e.g. when a process blocks to wait for something.

4.6.1.1 Context switching

Context switching, means switching from executing one block of code b_1 to another block of code b_2 with the intent to eventually return to b_1 in a way that is transparent to the blocks of code. This allows code to be partitioned into separate independent blocks that can run concurrently. This is opposed to monolithic blocks of code that entirely control the flow of execution. Therefore, context switching is essential for any operating system that wishes to utilise a scheduler to allow concurrent execution of code. Context switching works by saving the state of the CPU registers from one context to memory, and then switching to another context by restoring the state of the registers previously saved in memory. Since context switching must manipulate registers directly and individually, it is a feature that must be implemented on a very low abstraction level, and it can change significantly from one CPU architecture to another. The scope of this project is to only implement an operating system for a Cortex M4. Hence context switching needs only to work on the architecture used by this chip.

Context Switching on the Cortex M4 is partially handled by the Nested Vectored Interrupt Controller (NVIC). When an exception (this includes interrupts) occurs, the NVIC changes context to an Interrupt Service Routine (ISR). Upon changing context from application code to the ISR, the NVIC saves the registers R0-R3, R12,

LR, PC and PSR on the stack. These are only some of the registers. When the ISR exits, and control is handed back to the NVIC, the NVIC will restore these registers again.

If the operating system wishes to switch context, it must save the remaining registers, R4-R11 and stack pointer, to memory; pick a new context to restore; reload the registers R4-R11 and the stack pointer of the new context from memory, and hand back control to the NVIC. The NVIC will then reload the remaining registers from the stack of the new context. Table 4.1 shows the registers and who saves them.

Register	Saved by	Purpose
R0	Hardware	General purpose (Argument value + return value)
R1-R3	Hardware	General purpose (Argument value)
R4-R11	Software	General purpose (Local variable)
R12	Hardware	General purpose (Intra-Procedure-call scratch)
R13 (SP)	Software	Stack pointer [Banked]
MSP	Software	Master stack pointer (Stack pointer for kernel space)
PSP	Software	Process stack pointer (Stack pointer for user space)
R14 (LR)	Hardware	Link register
R15 (PC)	Hardware	Program Counter
xPSR	Hardware	Special-purpose Program Status Register

Table 4.1: A table of the CPU registers of a Cortex M4. Notice that register R13 (stack pointer) is banked; there are separate registers for user space and kernel space. Aliases are written in parenthesis in the Registers column, and calling convention is written in parenthesis in the Purpose column.

Additionally, the Cortex M4 utilises a banked[3] stack pointer. The logical SP register consists of two physical registers: the Master Stack Pointer (MSP) and the Process Stack Pointer (PSP). The MSP is intended for kernel space execution while the PSP is intended for user space execution, although this not enforce by hardware. Selecting which stack pointer to use can not be done by software within an ISR. Instead, this must be done by the NVIC. This makes it impossible to restore a context entirely from software. Software can instruct the NVIC which stack pointer should be used by the new context. Software can do this by putting a specific value into the PC register. This value is the EXC_RETURN value. Table 4.2 shows the different EXC_RETURN values and their meaning. An ISR is always concluded by putting an EXC_RETURN value in to the PC register. Before entering an ISR, the NVIC puts the EXC_RETURN value that denotes the state of the CPU before the interrupt was triggered, in to the LR register. When an ISR is not used to perform

[3]A banked register is a logical register implemented across multiple physical registers. The state of the hardware determines which physical register the logical register refers to[5]

a context switch, it concludes by putting the EXC_RETURN from the LR register in to the PC register. As this is custom for all function calls, this can be done by the compiler. When an ISR is used for context switching, the EXC_RETURN denoting the state of previous context may be different from the state of the next context. Therefore, the compiler is not be allowed to automatically conclude the ISR by putting the value of the LR register in to the PC register.
Additionally, for restoring contexts, the kernel records the EXC_RETURN value from the LR register.

Value	Description
0xFFFF FFF1	Return to Handler mode, using the Master Stack Pointer, but without the floating point unit.
0xFFFF FFF9	Return to Thread mode, using the Master Stack Pointer, but without the floating point unit.
0xFFFF FFFD	Return to Thread mode, using the Process Stack Pointer, but without the floating point unit.
0xFFFF FFE1	Return to Handler mode, using the Master Stack Pointer, with the floating point unit.
0xFFFF FFE9	Return to Thread mode, using the Master Stack Pointer, with the floating point unit.
0xFFFF FFED	Return to Thread mode, using the Process Stack Pointer, with the floating point unit.

Table 4.2: A table of the possible EXC_RETURN values. An interrupt service routine is exited by loading one of these values into the Program Counter register.

4.6.2 Interpartition communication

Interpartition communication is a basic mechanism for linking partitions by messages using channels. Each partition can be linked to multiple channels by ports. A partition can interact with its ports by using the APEX to send or receive messages.

At the application level messages are atomic and as such channels are required to ensure the correctness of every message received[34]. Multiple different modes exist for the messages and different checks and methods should be applied to comply with the ARINC 653 standard. Only a subset of these features is designed and implemented, to give a basic feature set which allows partitions to communicate on a single microcontroller.

A simplified design of queuing ports and sampling ports is made for Open653.

Since ports and their attributes are declared and defined statically, very little has

to be done to initialise them at startup. Ports are organised as belonging to some partition and channel. They have attributes according to their port type, as well as common attributes:

- The port name
- A flag indicating the port type
- A port direction indicating whether the port is a destination or a source
- Maximum message size
- A reference to the channel it belongs to

A port belongs to a partition and contains a reference to its channel. Some necessary attributes for queuing ports and sampling ports are listed in the sections 4.6.2.1 and 4.6.2.2 respectively.

4.6.2.1 Queuing Ports

The necessary attributes for queuing ports are the following:

- Number of messages received in the queue
- Maximum number of messages to be held by the receive buffer
- A circular buffer to act as the FIFO buffer

The OS has to facilitate some methods to push and pop complete messages to and from a circular buffer, which would be auto-generated from the configuration file. Open653 also has to provide functions to handle the following APEX calls defined in the standard[35]:

- CREATE_QUEUING_PORT
- SEND_QUEUING_MESSAGE
- RECEIVE_QUEUING_MESSAGE
- GET_QUEUING_PORT_ID
- GET_QUEUING_PORT_STATUS

4.6.2.2 Sampling Ports

The attributes for sampling ports are the following:

- A refresh period time
- An indicator for the validity of the last received message
- A buffer to hold the sampling message

The OS has to facilitate some methods to transmit and validate sampling messages. A simple buffer has to be provided for this by the auto-generated structures. Open653 also has to provide functions to handle the following APEX calls defined in the standard[35]:

- CREATE_SAMPLING_PORT
- WRITE_SAMPLING_MESSAGE
- READ_SAMPLING_MESSAGE
- GET_SAMPLING_PORT_ID
- GET_SAMPLING_PORT_STATUS

4.6.3 Memory Management

ARINC 653 advertises time and space separation covered in section 2.3. To accommodate for the feature of separating partitions in space, this operating system relies on the memory protection unit (MPU) to manage permissions across sections of memory, ensuring that partitions stay within a dedicated memory space.

Dynamic memory allocation is not permitted at runtime outside predefined memory sections. This policy ensures a fixed and predictable sandboxed environment, where partitions can not restrict other parts of the system by occupying memory resources. It also gives the system developer a way to ensure that the system has enough memory for all processes to function correctly, given that this amount is define in the configuration file.
Memory regions are declared in the configuration file and included at compile time, as a list of different system features, like individual partitions and communication buffers. The rest of this section will deal with the subject of managing memory regions on a STM32F415 chip and categorising and analysing the memory requirements into a single memory strategy to implement.

4.6.4 The MPU on a STM32F415

The MPU can be used to prohibit partitions from corrupting data used by kernel code and other partitions. The MPU can protect up to eight memory regions and is unified, meaning that it does not distinguish between code and data sections. Regions can have sub-regions, but because the minimum regions/sub-regions size is the length of a cache line (32 bytes), sub-regions are only available for regions of at least 256 bytes of size. Partitions above this threshold can have eight sub-regions.

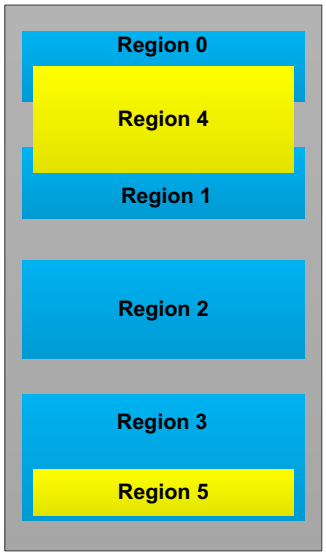


Figure 4.4: Example of nested and overlapping regions from [55].

Every region has a fixed priority and is numbered from 0-7, where region 7 has the highest priority. Regions can be nested and overlapping. The behavior when nesting and overlapping is dictated by the priority of the region as depicted in figure 4.4.

4.6.5 Memory strategy

As the memory available is not abundant, there is the need of avoiding wasting memory. For that, the memory needed by the partitions is optimised by a sorting algorithm in order to fit the most partitions possible in a limited region of memory and by expanding the partition size, fulfilling the empty gaps. It is important to keep the size, initial pointer and the region of each partition for enabling and disabling the right area of memory when a different partition is active.

4.6.6 System calls

To improve security and robustness, many modern operating systems restrict application code from accessing certain peripheral hardware (e.g. storage devices, I/O devices) as well as the control registers of the CPU. To take advantage of this, many modern CPUs utilise two or more execution levels; each level adding restrictions. The kernel runs in the most privileged execution level while application code runs in a more restricted execution level. This prevents application code from having direct access to privileged hardware. Application code must instead access privileged hardware through the kernel. For this, kernels often provide a fixed set of functions that application code may call. Calls to these functions are called system calls. System calls must escape the unprivileged application level and enter into the privileged kernel. This is accomplished by triggering an interrupt that the kernel will catch. By preparing a function designator (often just an identification number) and the call arguments on the stack and in the general purpose registers before triggering the interrupt, application code can signal its intent to the kernel.

System calls are critical to accomplish ARINC 653's strict space and time separation. Therefore, ARINC 653 explicitly specify that partition code must be executed in unprivileged mode only[36]. If partition code is not run in an unprivileged mode, it will have the ability to disrupt the system (e.g. changing the scheduling table and memory protection).

4.6.7 Error Handling

The error handler is designed to give the developers the possibility to see that something has gone wrong in the system. Since the health monitor is not going to be implemented, this is the extend to which the error handler is designed. The idea is to capture only some of the faults or failures and throw an exception that the error handler would then catch.

4.7 Schema

An ARINC 653 schema specifies the structure of the configuration and the required elements for the core module. The schema is a recipe used to create a configuration file.

4.7.1 Schema design

The schema defined in figure 4.5 is not the complete ARINC 653 schema, though it can be extended into the full schema with some more work. The scope was limited as outlined in the problem statement Problem statement.

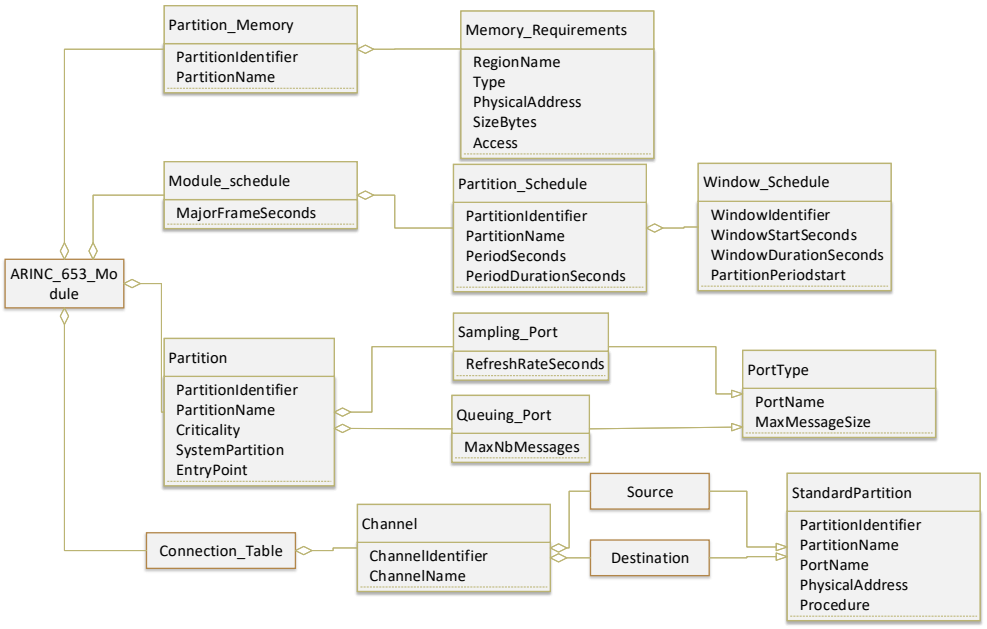


Figure 4.5: Open653 schema design

The figure presents the schema and the relationship between the elements. An ARINC 653 Module has four elements. One or more Partition Memory elements, a Module Schedule, one or more Partitions and a Connection Table.

A Partition_Memory has one or more Memory_Requirements and two attributes which correlate to its associated partition. The Memory_Requirements specify the configuration of Partition_Memory; the configuration includes the amount of memory it requires, the type of data stored, how the data is accessed and its physical address on the hardware.

A Module_Schedule has one or more Partition_Schedules and one attribute which defines the major frame. The major frame is the smallest amount of time possible to allow every partition to run at least once.

Partition_Schedule has one or more Window_Schedule and four attributes which correlate to its associated partition, and defines some timing details not used in this project. PeriodSeconds and PeriodDurationSeconds specify the maximum time a partition can use in one cycle for its Window_Schedule to complete. These two attributes are unused currently, as there is not appropriate error handling to make use of it and Window_Schedule attributes are sufficient for scheduling.

Window_Schedule defines when a partition needs to run and for how long it will run. PartitionPeriodStart is not used, and it purpose is not understood.

Partition has one or more queuing ports and one or more sampling ports; it has five attributes which specify information about the partition. EntryPoint is the name of the partition initialisation function, SystemPartition specifies what type the partition is. Criticality is not currently used, it is meant to define the importance of a partition.

The ports have four attributes which define information about the port such as size and source or destination. Sampling_Ports are not well supported currently in the project but will function. This is due to not having or needing any sampling ports in the current implementation.

A Connection_Table has one or more Channels and zero attributes. A Channel has one or more source and destination channels and two attributes which identify the channel. The Source and Destination have four attributes which correlate to its associated partition, and define the physical address of the channel. Open653 is missing an attribute called procedure from Channel, which is defined in the original ARINC 653 schema. Procedure is meant to be used by ports to interact with drivers.

4.7.2 Full ARINC 653 schema

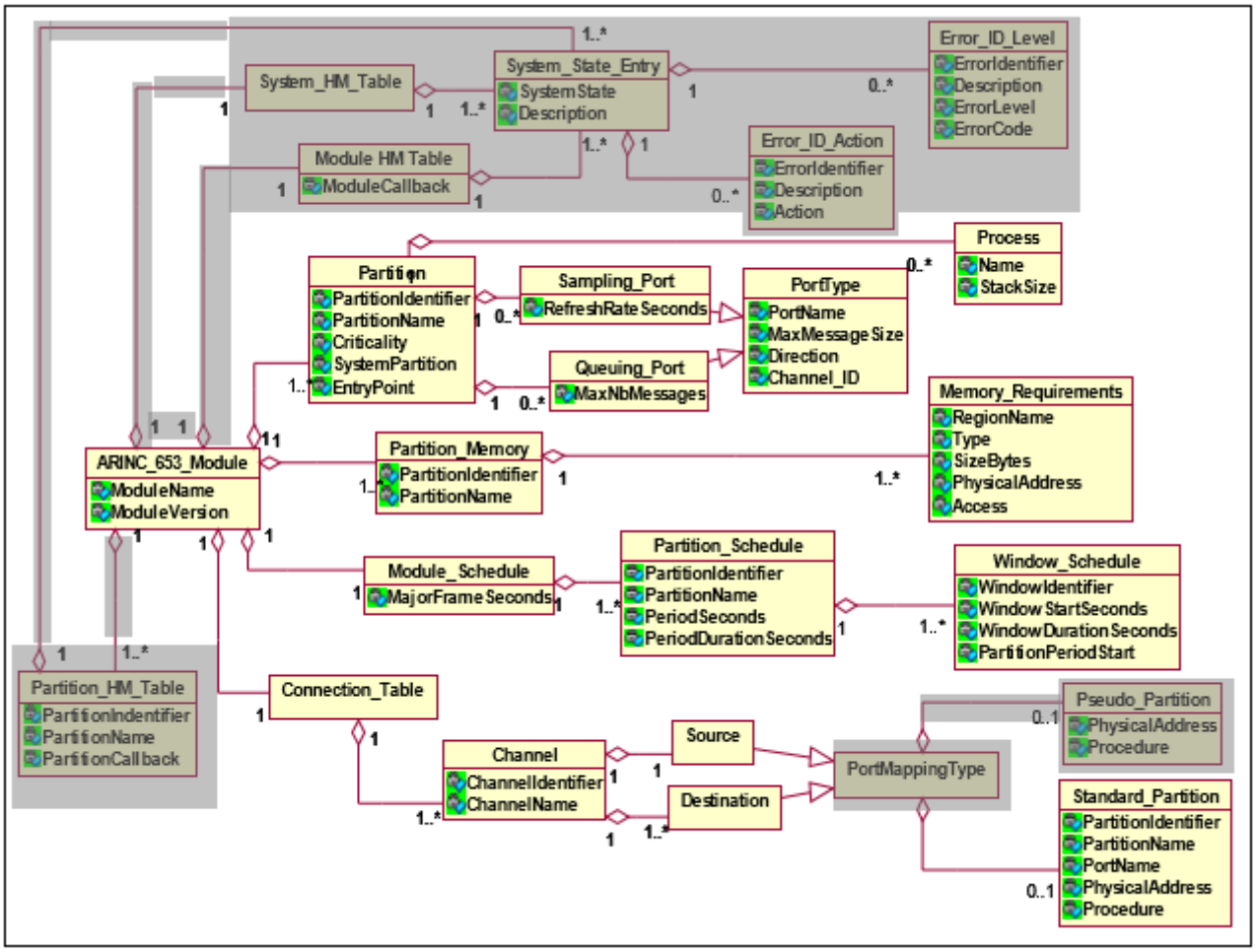


Figure 4.6: full ARINC 653 schema

The figure shows the intended schema for a fully compliant ARINC 653 implementation as found in the ARINC 653 standard. The greyed out areas, are the aspects of the Open653 schema not yet implemented in this project.

4.7.3 Schema translation

XML is the chosen configuration file format. XML was chosen as opposed to alternatives due to its easy to read and write nature, ARINC 653's XML example providing guidance and previous experience with XML in the project group.

4.8 APEX

The APEX defines an API for applications running on Open653. Its functions are grouped into seven major categories[37]. Table 4.3 contains the headers declaring these calls (column on the right), classified according to the standard.

Category	Header file
Partition management	apex_partition.h
Process Management	apex_process.h
Time management	apex_time.h
Memory management	*No APEX calls*
Interpartition communication	apex_sampling.h apex_queuing.h
Intrapartition communication	apex_buffer.h apex_blackboard.h apex_semaphore.h apex_event.h
Health monitoring	apex_error.h

Table 4.3: APEX calls

These header files only declare the calls, which will be implemented later in the kernel. *apex_types.h* declares their arguments: global and constant variables, as well as the types used by the calls. All the content is found in the standard, in the form of an ANSI C compliant code. The input (IN) arguments are types defined here. The output (OUT) arguments are dependent on the `RETURN_CODE` value. This can be one of the values shown in listing 4.1.

```
67 typedef
68     enum {
69         NO_ERROR        = 0, /* request valid and operation performed    */
70         NO_ACTION       = 1, /* status of system unaffected by request   */
71         NOT_AVAILABLE   = 2, /* resource required by request unavailable */
72         INVALID_PARAM   = 3, /* invalid parameter specified in request   */
73         INVALID_CONFIG  = 4, /* parameter incompatible with configuration*/
74         INVALID_MODE    = 5, /* request incompatible with current mode   */
75         TIMED_OUT       = 6  /* time-out tied up with request has expired*/
76     } RETURN_CODE_TYPE;
```

Listing 4.1: *apex_types.h*

4.9 Partitions and processes

Partitions as described in section 2.3.1 are defined at compile-time and contain one or more processes, which are initialised during run-time. A partition is defined to be one or more structures containing information about its execution and resources. As described in the Schema section, there are multiple groups of information about each partition. It makes sense to separate this information for different parts of the system (e.g. only the scheduler needs to know about a partitions' execution window information).

While information about the overall partition is defined statically, its processes are defined and created at runtime and for this reason every partition needs some memory allocated to store this information. A starting process is created by default using the partitions' entry-point, which is then allowed to spawn any additional processes.

Processes, just like partitions are collections of information. Processes hold information about their own state and how they are to be scheduled.

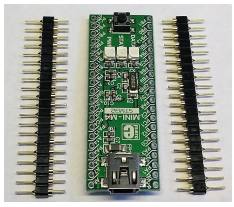
CHAPTER 5

System Implementation

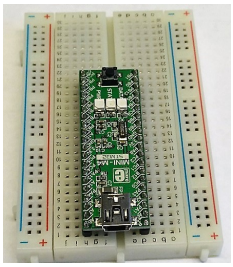
This chapter deals with how the system's functionalities are implemented. Diagrams are included when necessary, as well as code snippets for the interesting parts of the program.

5.1 Hardware

The Mini-M4 board is a development board, which means that beside the microcontroller, it also contains two crystal oscillators, two LEDs, a reset button and the peripherals for regulating the power it gets from the micro-USB connector. Some of the GPIO connections can be accessed using external pins. These pins are soldered to the board, and this is setup on a breadboard for practicality, as seen in 5.1.



(a) MINI-M4 before soldering



(b) MINI-M4 on a breadboard

Figure 5.1: Preparation of the board

The JTAG connector is wired to the five corresponding pins of the board. The serial

communication is established with two other wires, going to the UART pins of the board. The final setup can be seen in the next figures.

(a) The JTAG lines

(b) Both the JTAG and serial interfaces

Figure 5.2: Connecting the board

After this, the board is ready to be used in the development process. The only drawback of this setup, is that it requires three USB ports: one for powering the board, another one for the JTAG interface, and a last for the serial communication. For detailed photos of the process, consult Appendix B.

5.1.1 JTAG Interface

Figure 5.2a represents the physical layout of the JTAG interface. Since the JTAG connector has 20 pins, and only some of these are required to communicate with the board, the set up process took some time to figure out. The following diagram, shows the pinout of the JTAG and the corresponding pins on the microcontroller's JTAG port.

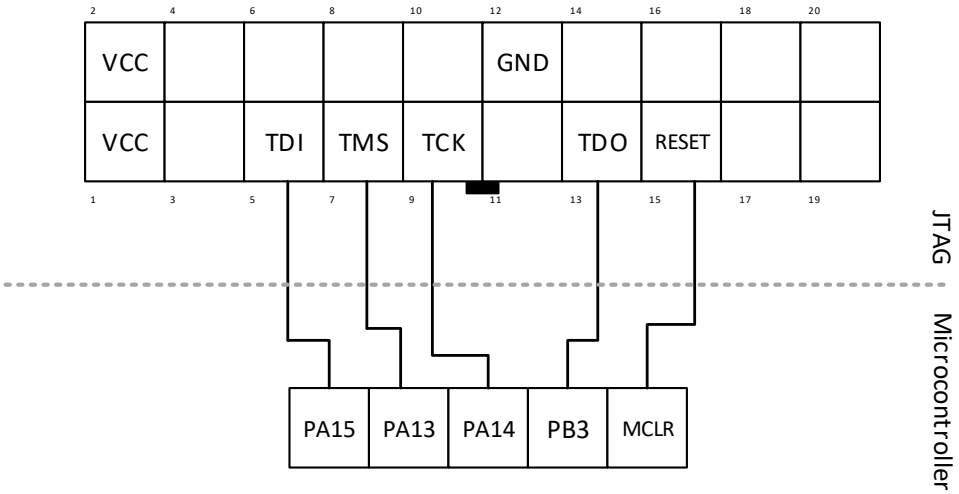


Figure 5.3: The connection of the JTAG interface

The tags on the upper part of figure 5.3, are JTAG specific:

TDI Test Data In pin

TMS Test Mode State pin

TCK Test Clock pin

TDO Test Data Out pin

RESET used for resetting the target device. This is connected to the MCLR (Master Clear or Reset) pin of the board

The tags on the lower part of the figure represent the pins of the microcontroller, where *P* stands for port, *A* or *B* for the port identity (*PORTA* or *PORTB*) and the number being the individual GPIO needed.
A driver[1] is needed to be installed on Windows machines, in order to recognize and program using ST-LINK/v2.

5.2 Source Structure

The following sections describes different aspects the software used and the source code located in the project root directory (ESS7_project) denoted as '/'. Below the root folder, all the source code and compiled binaries are structured as listed in figure 5.4:

[1] STSW-LINK009 v1.02

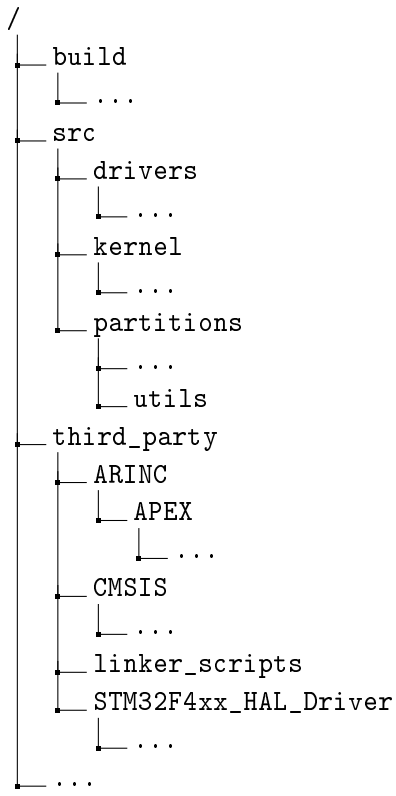


Figure 5.4: A rough depiction of the source file structure

5.3 Toolchain

To write, compile and load code to the embedded platform, a regular computer is used with the editor of choice. A C compiler and a programmer unit is used to transmit the code by the JTAG interface (5.1.1). The toolchain is adopted from AAURacing[1] which uses a similar chip and develop on Linux/OSX. For this project the toolchain is generalized to also work on Windows 10. In the rest of the section the toolchain and source code structure are explained.

5.3.1 CMAKE

CMake is used to setup the makefiles for the project. It is of little benefit comparing to writing the makefiles from scratch, but has been adopted in Open653, since it came with the toolchain.

> "CMake is an open-source, cross-platform family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent

> configuration files, and generate native makefiles and workspaces that can be used in the compiler environment of your choice[9]".

Like Make, CMake is run from a file containing a set of instructions on a project. The instructions are separated in a nested structure of files to limit complexity. A project can have multiple CMake files, they all have to be named *CMakeLists.txt* and hence are placed in separate folders. The main CMake file is placed in the root-folder of the project and contains the main compiler information. Specific information about compile targets and their files are placed in sub files in folders as depicted in figure 5.5.

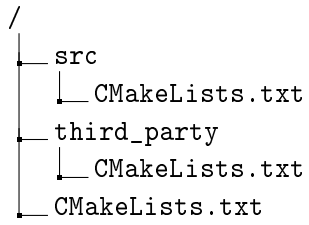


Figure 5.5: Placement of CMakeLists.txt files in source directory.

As denoted by the first line of the CMakeLists.txt file in '/', no less than version 2.8.8 of CMake is required. The rest of the file contains:

- the compiler to use
- a list of compiler flags (section 5.3.2.1)
- commonly used include paths
- two required defines (section 5.3.2.1)
- a command to flash the target device using OpenOCD (section 5.3.3)

Using CMake results in a tree of makefiles, which can be compiled by running Make from within a build folder. A complete tutorial on how to get started using the build system can be found in Appendix C.

Make is a UNIX tool[18] often used to make many calls to GCC in the C language projects. Objects are compiled in separate calls to GCC and Make provides features to set up a dependency tree of calls when many objects are linked together in a target.

5.3.2 GCC

The GNU Compiler Collection or GCC is used for compiling and linking the C files into a target called 'OS'. In order to compile for the ARM architecture a cross compiled version of GCC is necessary, in this case the *arm-none-eabi-gcc* compiler which is available in all newer Ubuntu versions[65].

5.3.2.1 Compiler Flags

The compiler relies on a list of flags to compile the code to the programmers liking. This sub section will cover the necessary or otherwise important flags for compiling Open653. The entries table 5.1 can be found in the main CMakeLists.txt as referred to in section 5.3.1.

MCU flags

These options can be found at gcc.gnu.org [14].

-mcpu=cortex-m4	This specifies the name of the target ARM processor.
-mtune=cortex-m4	This option specifies the name of the target ARM processor for which GCC should tune the performance of the code.
-mthumb	Generate code that executes in Thumb state.
-mlittle-endian	Generate code for a processor running in little-endian mode.
-mfpu=fpv4-sp-d16	This specifies what floating-point hardware is available on the target.
-mfloat-abi=hard	Specifies which floating-point ABI to use. 'hard' allows generation of FPU-specific floating-point instructions.
-mthumb-interwork	Generate code that supports calling between the ARM and Thumb instruction sets.

Linker flags

These options can be found at gcc.gnu.org [17].

-Wl, -T.../file.ld	Use the linker-script STM32F415RG_FLASH.ld in the folder /third_party/linker_scipts (section 5.3.2.2).
-nostartfiles	Do not use the standard system startup files when linking. Used to force GCC to use a custom implementation of Newlibc implemented functions.
-Map=./result.map	Print a link map to the file result.map showing where object files and symbols are mapped into memory.

Debugging flags

These options can be found at gcc.gnu.org [16].

-g	Produce debugging information.
-ggdb	Produce debugging information for use by GDB.

C flags

These options can be found at gcc.gnu.org [15].

-std=c99	Determine the language standard to be ISO C99.
-fms-extensions	Accept some non-standard constructs, useful in some static structures (section 5.8.3).

Table 5.1: Showing the important GCC flags used to compile the OS.

There are two important flags, which have not been mentioned in table 5.1. the `-DHSE_VALUE=16000000` and `-DSTM32F415xx` are not compiler options but definitions passed to GCC to use at compile time.

`-DHSE_VALUE=16000000` defines the value of the external oscillator in hertz. Specifically it defines `HSE_VALUE` to be 16 million hertz as is specified in [46]. The input of the external oscillator is used as the source for the main clock (section 5.4.1).

`-DSTM32F415xx` simply adds `STM32F415xx` as a definition at compile-time. It is used by the HAL library to determine which processor to set up for.

5.3.2.2 Linker Script

A linker script is used by the compiler to determine where to place different sections of the compiled binary. Because of this, everything in the output binary gets an offset that matches the memory layout described in section 4.3. The file `result.map` (table 5.1) can be used post compilation to determine the location of all symbols in the binary, as a way to debug the linker-script.

The linker-file STM32F415RG_FLASH.ld is part of the toolchain and is simple to edit to fit any member of the STM32F4 family. In this project the only necessary change is to specify that the flash is 1 MB. Beyond that, the changes depend on the partition requirements of the system.

Listing 5.1 shows how the memory layout is defined in the linker script. While `FLASH`, `RAM` and `CCMRAM` define mappings of memory in hardware (see section 4.3), `OS`, `DUMMY1`, `DUMMY2` and `STDIO_SYS` define the location and sizes of the system memory. In listing 5.1, `ORIGIN` denotes the starting address of a section of memory and `LENGTH` denotes the size of the section.

```
40 /* Specify the memory areas */
41 MEMORY
42 {
43   FLASH (rx)      : ORIGIN = 0x08000000, LENGTH = 1024K
44   OS (rx)         : ORIGIN = 0x08000000, LENGTH =   64K
45   DUMMY1 (rx)     : ORIGIN = 0x08010000, LENGTH =    8K
46   DUMMY2 (rx)     : ORIGIN = 0x08012000, LENGTH =    8K
47   STDIO_SYS (rx)  : ORIGIN = 0x08014000, LENGTH =    8K
48   RAM (xrw)       : ORIGIN = 0x20000000, LENGTH =  128K
49   CCMRAM (rw)     : ORIGIN = 0x10000000, LENGTH =   64K
50 }
```

Listing 5.1: *third_party/linker_scripts/STM32F415RG_FLASH.ld*

By default all symbols and associated memory goes into the `OS`. Only when a partition is specifically defined to a location in the linker-script, can it be separated from the `OS`. Listing 5.2 shows how to allocate a specific partition. The linker-script uses

the fact that every partition is compiled as a library, and the library name is added in the listed format under its corresponding label (in this example: `STDIO_SYS`). This scheme assures the code is separated from the kernel in flash. It does not, however, separate the statically allocated variables and structures in RAM.

```
85 .stdio_sys :
86 {
87   . = ALIGN(4);
88   libstdio_sys.a:*
89   . = ALIGN(4);
90 } >STDIO_SYS
```

Listing 5.2: *third_party/linker_scripts/STM32F415RG_FLASH.ld*

Similar to listing 5.2, the compiled libraries of the compiled partitions are confined to a space in flash, as defined by their labels in 5.1.

5.3.3 OpenOCD

Open On-Chip Debugger or OpenOCD[51] is used together with the ST-LINK/v2 to program the chip and create a channel for GDB to debug the system at run-time. The CMakeLists.txt file in '/' defines a function running the following commands using OpenOCD:

- `init` - Initialize connection
- `reset halt` - Reset and halt the system
- `flash write_image erase ${elf_file}` - Write compiled output binary image to chip
- `reset run` - Reset chip and start normal execution

After execution of this command, OpenOCD will remain open and connected to the chip and will accept incoming connections on port 3333 for GDB debugging as a GDB server (see more in 5.3.4).

5.3.4 GDB

The GNU Project debugger[12], GDB (*arm-none-eabi-gdb*), is an open source software debugger built to compliment the GNU Project compilers, GCC (see 5.3.2). It allows programmers to debug their applications. GDB can:

- attach to a remote program, even running on a different piece of hardware
- pause execution of a program at specified lines of code with breakpoints

- show the content of memory, either by direct addressing or by variable name
- if the hardware supports it, show the value of each CPU register
- execute code line by line, letting the programmer see how each line affects the state of the program, and in case of crashing code, which line of code crashes the program

This feature set, makes GDB especially useful for tracking down complex bugs and bugs in otherwise inaccessible systems, (e.g. systems with no screen to directly print to). GDB also allows programmers to investigate buggy code while it is running, as opposed to after-the-fact debugging of logging, printing, etc. Many advanced text editors and integrated development environments feature direct seamless GDB integration. Outside of such programs, GDB can be run in a terminal.
To debug the code running on the chip, an active OpenOCD connection to the chip is required. When OpenOCD is connected, it will act as a GDB server. GDB can be connected to OpenOCD with the command `target remote localhost:3333`.

5.4 Drivers

Setting up the drivers took place in the early phases of the project. Their functionality rely a lot on the HAL library. The library's documentation is a good starting point, and the things it lacks can be found in the microcontroller's datasheet.

5.4.1 System clock

The initial setting for the clock is set to use the internal oscillator, which runs at 16MHz. In order to get the maximum speed supported by the CPU, the board needs to be configured to multiply this frequency. However, instead of using the internal oscillator, which can be imprecise, the Mini-M4 board has an external oscillator[2] that is used as the main clock source. This is represented in figure 5.6 with the HSE tag. These tags refer to the location and the speed of the oscillators present on the board or inside the MCU:

HSE high-speed external oscillator, the main clock source

HSI high-speed internal oscillator, not used

LSE low-speed external oscillator, used by the real-time clock

LSI low-speed internal oscillator, used by the independent watchdog timer

[2]Featuring less wander (slow variations due to temperature and aging) and jitter(fast variations)

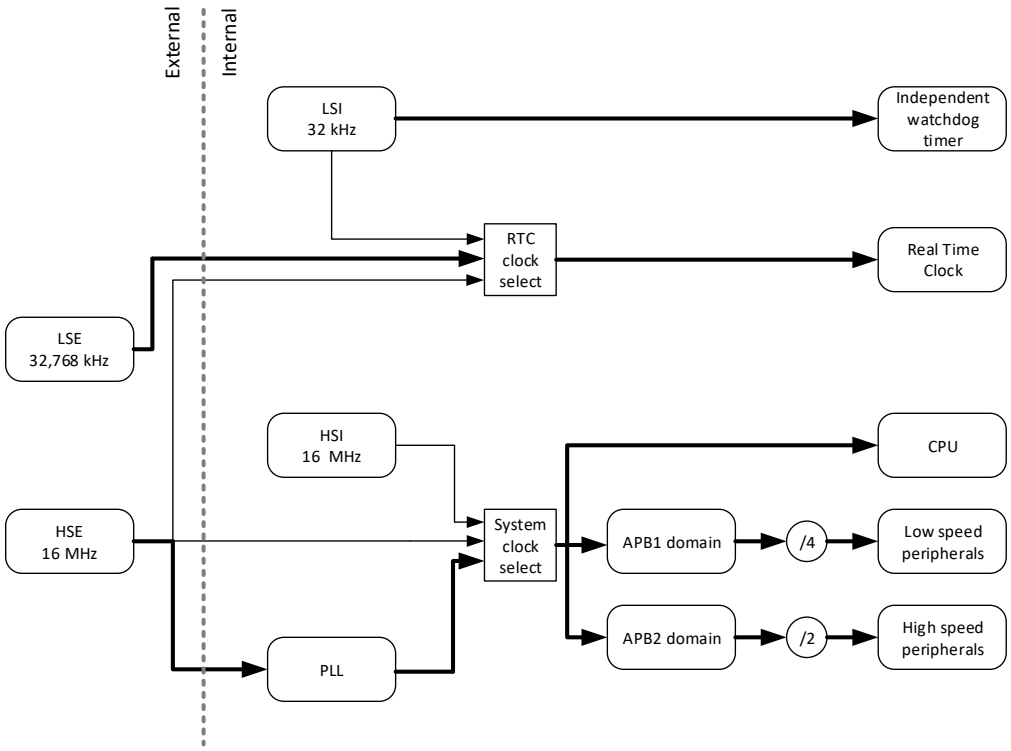


Figure 5.6: System clock overview [57]

The lower part of figure 5.6 shows that the frequency of the HSE is multiplied by a PLL (Phase-locked loop) system, before it gets to the clock selection block. This is the frequency supplied to the CPU and memory through the AHB bus, and scaled down for the peripheral buses. These buses are divided into low and high speed domains as:

AHB Advanced High-speed bus. This runs at 168 MHz, and connects mainly the CPU, memory and the GPIOs

APB1 domain low-speed domain, running at 42 MHz. This includes among other peripherals the UART

APB2 domain high-speed domain, running at 84 MHz

The PLL registers[3] have to be set up in such a way that the frequency it generates is compatible with the prescaling settings of the peripheral buses.

[3]Specifically the division/multiplication factor registers

5.4.2 UART

The UART allows the transmission and reception of information between the MINI-M4 and a terminal. Information is transceived one character at a time. An UART has a fractional baud rate generator system and the speed is limited by the GPIO frequency. UART4 is used with pin 10 and 11.

5.4.3 LEDs

The LEDs are used mainly for debugging. There are three LEDs on the Mini-M4 board, two of them being programmable. Since they are connected to two pins of a pins, these need to be initialised first. Afterwards, using the functions defined in the HAL library, these pins can be turned high, low or toggled, thus changing the state of the LEDs.

5.4.4 Delay

The delay function is also used for debugging, giving the developers the chance of seeing *when things are happening*. This is because most of the program instructions are executed so fast, that the they appear to be instant. Delay is defined in the HAL library, and relies on the SysTick.
SysTick is a timer[4] that can be used to generate the main system event clock. That is exactly how the HAL library has it implemented it. Its default time base is one millisecond. This means that every millisecond, SysTick generates an interrupt (to look after sensor values, scheduling or basic timing tasks). To see a more detailed explanation of the SysTick timer consult 5.8.2.1.

5.4.5 MPU

The MPU functionality of the MCU was tested but not implemented, beyond the extend of getting the driver working.

5.4.6 Timing

This driver is used to accurately measure the execution time of different functions. It uses the DWT (Data Watchpoint and Trace support) internal peripheral registers defined in the ARM-M4 Architecture manual[6]. Since the HAL library does not define them, they are accessed by their addresses. These addresses are found in the range shown in the upper part of figure 4.2, while the registers used by this driver are shown in listing 5.3.

[4]Fed with the system clock optionally divided by 8

```
 9 volatile unsigned int *DWT_CYCCNT  = (unsigned int *)0xE0001004;
10 volatile unsigned int *DWT_CONTROL = (unsigned int *)0xE0001000;
11 volatile unsigned int *SCB_DEMCR   = (unsigned int *)0xE000EDFC;
```

Listing 5.3: *src/drivers/time_get.c*

DWT_CYCCNT is a 32 bit counter, that when enabled counts the number of core cycles. *DWT_CONTROL*[5] is the control register for the Data Watchpoint and Trace support. By setting its least significant bit, the cycle counter mentioned above is enabled. The last register, *SCB_DEMCR* (Debug Exception and Monitor Control Register), and is used to manage exception behavior during debugging.

At a core frequency of 168 MHz, *DWT_CYCCNT* is incremented every 5.45 nanoseconds. In order to convert the number of cycles into time, the function in listing 5.4 was used.

```
33 int64_t convert_cycles_to_time(uint32_t cycles)
34 {
35     return (int64_t)((cycles / 168) * 1000);
36 }
```

Listing 5.4: *src/drivers/time_get.c*

The returned value is a 64 bit signed integer, as the ARINC 653 standard specifies [38]. The count is incremented in the range of nanoseconds, but the sampling frequency limits the step to 5.45 nanoseconds. In order to fully comply with the standard (one nanosecond steps), A counter would be necessary with a 1 GHz clock source.

5.4.7 Watchdog timer

The MCU has two watchdog timers ready to reset the program, in case of an error. The first one, the independent watchdog timer is based on its dedicated oscillator [6]. This way, the the watchdog could catch a fault in the program, even if the other oscillators fail. Its basic functionality is represented in figure 5.7. After the program starts running, a function is called that initialises and starts the watchdog timer. Here, a period of time is defined, in which the program has to refresh (reload) the watchdog timer. If this fails to happen, either by mistake, the developer forgetting to call the refresh function, or by a software/hardware fault, the watchdog timer restarts the program (depicted with the red arrow).

[5] Referred to as DWT_CTRL in the ARM documentation

[6] LSI running at 32 kHz, seen in the upper part of 5.6

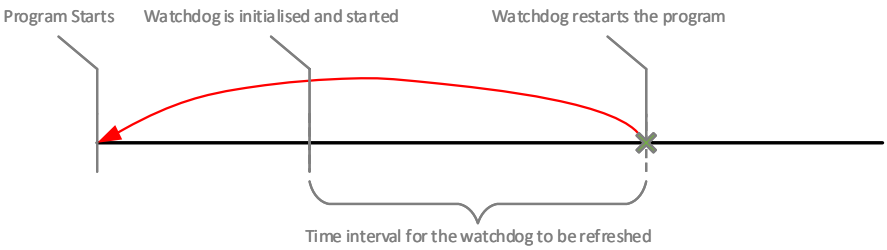


Figure 5.7: Basic watchdog timer functionality

The second watchdog timer, is called a window watchdog. In contrast to the independent watchdog, one can provide a time interval in which the timer has to be reset. Resetting the timer outside this interval (as well as not resetting it in the mentioned time window) would restart the program. It uses the system clock, which means that if this fails, the watchdog won't be able to reset the system. This watchdog has not been implemented, but its functionality is relevant to Open653, in regards to what the standard specifies. The processor should be able to isolate a partition that fails [22]. This could be done by using the window watchdog timer to define the time intervals for partitions to run. The feature called Early Wakeup Interrupt can be used to do some safety operations, just before the reset is done by the watchdog[58]. In short, if the window watchdog is not reset in the window interval it receives (correspondent to each partition), then the EWI will isolate the partition and the MCU will be reset to continue its operation with the remaining partitions.

5.4.8 RTC

The real-time clock is implemented as a way to give the system the possibility to keep track of time. It was realised at a later point that ARINC 653 specifies that time keeping should be done in nanoseconds, a range that the RTC is not capable of. This is the reason for continuing researching other ways of timing the system, as seen in section 5.4.6.

The RTC could be used for timestamps on error logs and such, as the standard specifies[39]. The use of the LSE oscillator, as well as the design of the RTC[7], makes this a reliable secondary clock source.

[7]Which includes all time components, as well as subseconds and year correction

5.5 XML configuration file

Writing the XML configuration file was an evolutionary process throughout the implementation phase. To begin with, the example XML provided by the ARINC 653 standard was used. As functionality within the system became available, the example XML values were modified for the desired purposes. All content in the XML was inputted manually, there is no tool support to modify the content.

5.6 XML Parser

The configuration of the OS is not usable in its XML format. Open653 is written in C; therefore a dynamic parser written in Python called *createCStructs.py* has been created to translate the XML and generate C structs. Using an external library *xmltodict.py*[69], the XML is converted into a JSON structure which makes navigating and accessing sub nodes easy. The generation of the of the structs are handled as two separate files; *xml_data.h* and *xml_data.c*.

5.6.1 xml_data.h

xml_data.h contains the static declarations for the structs written to the *xml_data.c*, and the symbols for the *xml_data.c* functions and variables.
The declarations for the structs are included into *xml_data.h* from a file called types.h. The symbols for the functions and variables are written to the file whilst parsing the XML.

5.6.2 xml_data.c

xml_data.c is mostly dynamic and generated from the elements in the XML. The static elements of its generation are handled by two functions and the self.idle_partition set in the init. The static functions are called write_file_c_header and write_file_c_footer. These functions write the static code needed at the beginning and the end of the *xml_data.c* file to function, e.g. the includes and the #endif. The idle partition is not included in the XML. It is hard-coded into the partition loop as it is always required to exist. For more information about the idle partition refer to section 5.9.3.

There are four dynamic elements (partitions, partition_memory, module_schedule and connection_table) which are acquired from the JSON structure. They are looped through, populating formatted strings, and are then written to *xml_data.c*.

5.7 C Structures

The data structure undergoes change during the translation from the XML described in section 4.5 to C structs to support ease of use. The structure of the data and relationships of the elements are presented in the figure 5.8.

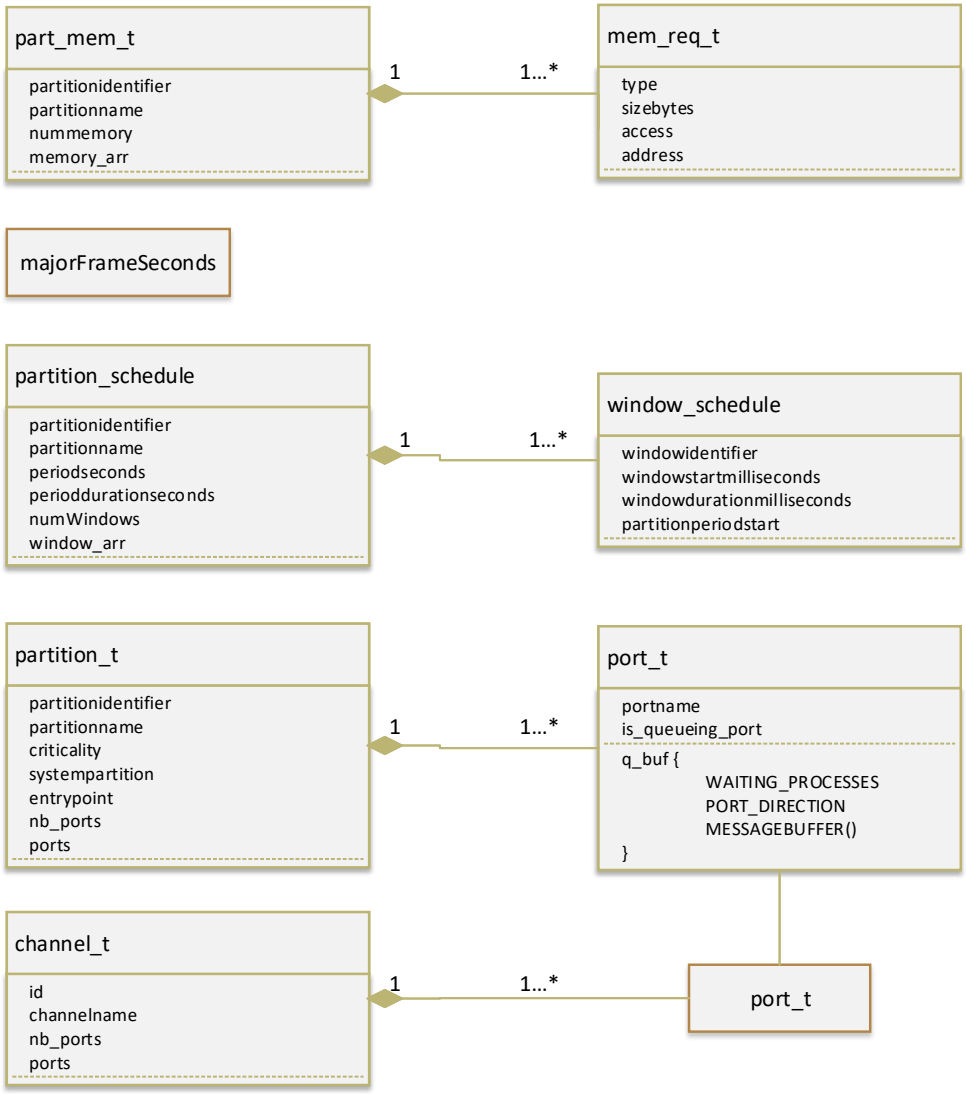


Figure 5.8: C structs design and relationships

The global structure of ARINC_653_Module (figure 4.5) has been removed, its sub elements now exist without a dependency. Module_schedule as also been removed, leaving its attribute majorFrameSeconds as an independent variable definition. partition_memory (part_mem_t) and partition_schedule have maintained their one-to-many structures with their sub elements. Partitions' sub elements, sampling and queuing ports, are combined into a single structure called ports. The XML element Connection_Table is removed, and StandardPartition is transformed into a port reference list which shares an association with the ports.

The following structures have been extended to contain an array of its substructures:

- part_mem_t contains an array memory_arr which references to the memory information of a given partition
- partition_schedule contains an array windows_arr which references to the timing window in which a partition is executed
- partitions_t contains an array ports which references to the ports available to the partition
- and channel_t contains an array ports which references to the ports available to the channel

Each of these structures has an associated variable to indicate the number of elements in the array.

port_t is a generic structure defining a port independent of its type as detailed in section 5.8.3.

5.8 OS kernel

This section gives an overview of the features of Open653 and how they are implemented. At the end a feature list is provided to give summary of the features and services implemented in Open653.

5.8.1 OS Initialisation

Open653 is initialised from the `main()` function in */src/kernel/main.c*. Its job is to set up the following:

1. The main clock
2. UART and clear the terminal

3. Red and yellow LED
4. The ports
5. Partitions and their processes
6. Construct a partition schedule
7. Enter unprivileged mode by setting a bit in a register

A more complete implementation of ARINC 653 would also have to set up the MPU. Though the MPU is considered in design (section 4.6.4), and testable code does exist, it is at this point yet to be included into the kernel as a service and for this reason no setup is done.

The setup of the UART is fixed to use port C pin 10 and 11 with a baud-rate of 115200 baud. Setting it up is a matter of passing these arguments to the HAL library and is not covered further in the report.

The last action taken by the main function is enabling the SysTick interrupt for starting the scheduler and entering unprivileged mode.

In the following sections the main clock setup (section 5.8.1.1), the partition and process setup (section 5.8.1.2), and partition schedule setup (section 5.8.1.3) are explained in further detail.

5.8.1.1 Main Clock Setup

The system clock is setup using its specific driver, explained in 5.4.1. The driver sets the core to run at 168 MHz, as well as scaling this frequency for the peripheral buses.

5.8.1.2 Partition and Process Setup

The idle partition (explained in section 5.9.3) is set up first using its entrypoint `idle_main`.

The rest of the partitions listed in the auto-generated C file are initialised by setting the number of active processes to one, setting up their specified entrypoint with a memory address within their memory space and setting the rest of the empty process entries to `DORMANT`.

Note that in this implementation every process is given 1 KB of RAM to use for stack. This can cause a fault if partitions initialise more processes than their defined memory spaces hold.

5.8.1.3 Building the schedule

The partition schedule is defined in the XML schema file as windows (see 5.8.2.3). Windows output by the parser, are ordered by partition, not in chronological order. Additionally, windows in the XML schema are not sub-elements of partitions, but are instead separate elements, referencing the partition they belong to by ID.

As a result, to do any scheduling of windows, it is necessary to search through all windows to find the next one within a major frame, and then search through all partitions to find the partition with the same ID. To make matters more complicated, the XML schema may contain implicit idle periods (periods with no defined windows). To lighten the burden on the scheduler, the work to build a schedule by ordering the windows in chronological order and inserting explicit idle periods are done during system initialisation.

Provided the same XML schema, the schedule is always the same; each time the system boots, a schedule identical to the one last used is built. In future builds of Open653, the schedule should ideally be built by the XML schema parser, to avoid rebuilding the same schedule during each boot of the system.
The algorithm to build the schedule is as follows:

```
   /* excluding implicit idle periods                                    */
 1 totalWindows ← number of windows in xml;
 2 desiredStartTime ← 0;
   /* This is just an arbitrary high number                              */
 3 earliestNextStartTime ← 5000;
 4 matchingWindow ← NULL;
 5 while windows in schedule < totalWindows do
 6     foreach window w in xml schema do
 7         if start time of w = desiredStartTime then
 8             matchingWindow ← w;
 9             break;
10         else if start time of w > desiredStartTime and earlier than
             earliestNextStartTime then
11             earliestNextStartTime ← start time of w;
12     if matching window is found then
13         partition;
14         foreach partition p do
15             if ID of p = ID of partition of matchingWindow then
16                 partition ← p;
17                 break;
18         insert window for partition into schedule;
19         desiredStartTime ← start time of w;
20     else
21         insert window starting at desiredStartTime and lasting
             earliestNextStartTime − desiredStartTime for idle partition into
             schedule;
           /* This is just an arbitrary high number                      */
22         earliestNextStartTime ← 5000;
23         totalWindows + +;
```

Provided that the data in the XML schema is valid, this algorithm will create a schedule of chronologically ordered windows, including windows pointing to the idle partition.
Figure 5.9 shows what the finished schedule may look like, given a certain schedule in the XML schema.

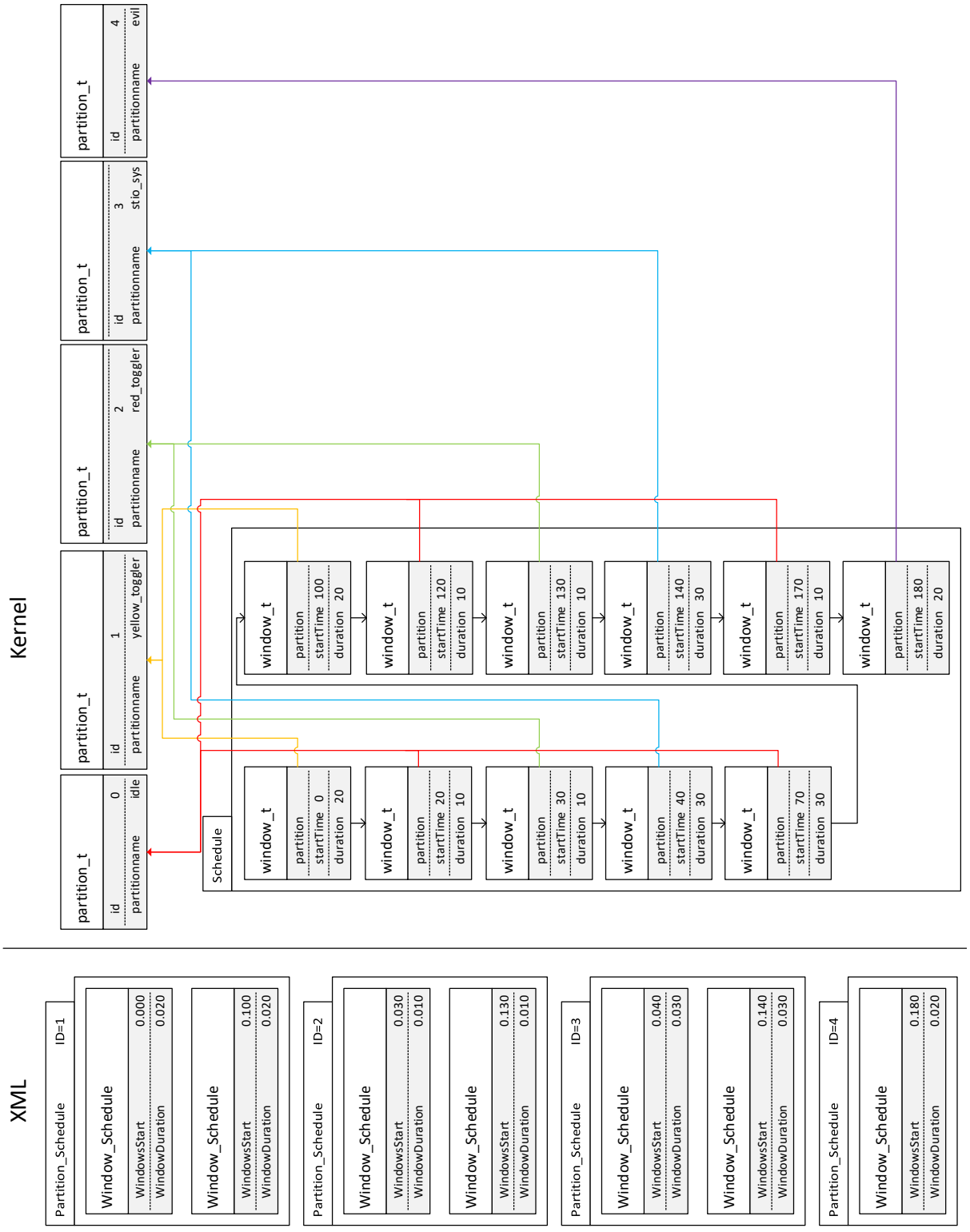


Figure 5.9: Windows as represented in the XML schema, and in the kernel after the schedule has been build

5.8.2 Scheduling

The ARINC 653 specification defines two levels of schedulers and the rules by which these schedulers decide which processes may execute at any given time. There are two different kinds of rescheduling, temporal rescheduling and event driven rescheduling. Partitions are exclusively rescheduled because of time, whereas processes may be rescheduled because of both time and events. As for temporal scheduling, the ARINC 653 specification does not explicitly specify a minimum or maximum time resolution for scheduling, though it does imply a resolution of 1 nanosecond should be available[38].

For Open653 the SysTick is used for triggering temporal scheduling. This led to a time resolution of 1 millisecond. Scheduling is never disabled; every millisecond the scheduler will be invoked and check which process of which partition should run next.

Scheduling is triggered within the function `SysTick_Handler`. This function is called directly by the NVIC when the SysTick timer triggers an interrupt. `SysTick_Handler` first saves the current context, then invokes the partition scheduler, followed by the process scheduler. Once the process scheduler finishes, the `SysTick_Handler` restores the context of chosen the process and returns control to the NVIC.

5.8.2.1 SysTick

The SysTick timer is a 24-bit decreasing timer that wraps when reaching zero. Unlike other timers on the MINI-M4, this timer is specified as part of the ARMv7M specification. Other timers are implementation specific; the vendor decides which timers to add to a chip. Therefore, between two different boards, there may be a different number of timers, and timers may have a different resolutions. This makes it more difficult to port code from one board to another. The SysTick timer however, is guaranteed to be 24-bit on any board that implements it. In the Cortex-M group of ARM microcontrollers, the SysTick timer is required for M3, M4 and M7; any code written for these can assume portability.

The clock source for the SysTick timer is the CPU clock, optionally divided by eight. The reset value of the SysTick timer is programmable. This makes it possible for software to program how often this timer wraps. Unless otherwise specified, this timer operates in polling mode; software has to poll for wraps, but it is possible to program the SysTick timer to trigger an interrupt upon wrapping.
By default, the HAL library sets up the SysTick timer to wrap every 1 millisecond and to trigger an interrupt upon wrapping.

5.8.2.2 Context Switching

There are multiple different states the processor can be in, when a context switch from one process to another occurs. The state indicates whether the Master Stack Pointer or the Process Stack Pointer was used by the process, and whether the FPU was used by the process or not. Because of this, the context switch algorithm must start by determining the processor state in order to know which registers to save and which stack to save them to.
The processor state is saved in the LR register, and it can have one of the possible six EXC_RETURN values (see table 4.2).
If the pre-empted process used the FPU, the FPU registers S16 to S31 should also be saved by software in addition to the general purpose registers. Because of time constraints, this was not implemented.
The registers are saved entirely on the stack. Therefore, it is important for the software to know which stack pointer to use when saving the remaining registers. Therefore the first thing the interrupt service routine (ISR) used for scheduling does, is to inspect the EXC_RETURN value in the LR register to determine whether to use the Master Stack Pointer or the Process Stack Pointer. Registers are saved at the address denoted by the stack pointer, before saving both the stack pointer and the EXC_RETURN value to the process structure elsewhere in memory. This is all done in assembly to make sure that there is complete control over which registers are used, so as to avoid overwriting any registers that have yet to be saved. This is absolutely necessary, as otherwise the state of register for a process might be corrupted during a context switch. Additionally, any ISR used for scheduling must be declared as naked[8], to avoid the compiler changing registers before the context can be saved, and to give complete control over which value is pushed to the PC register at the end of the ISR. Listing 5.5 shows the code that saves contexts on the stack.

After the context has been saved, control is handed over to the scheduler, which then decides which process should run next.
Once all the operations the ISR has completed, the context of the process selected by the scheduler must be restored. From this point, the code is once again written in assembly to avoid overwriting a register after it has been restored. First, the registers saved by software on the stack must be restored, then based on the EXC_RETURN value that the process interrupted with, it can be determined to which stack pointer register the new stack pointer must be pushed to. Lastly, execution must return to the process by pushing the EXC_RETURN value to the PC register. The EXC_RETURN value pushed to the PC register must be the same

[8] A naked function has no compiler generated prologue and epilogue. This means that the compiler does not save any registers on the stack before using them, and it does not restore them before ending a function.

```
37 __attribute__((always_inline))
38 static inline uint32_t context_save()
39 {
40     // Create a new variable to store the context-saved-stack pointer of
           the previous process.
41     // We explicitly choose the register to avoid GCC picking a register
           that hasn't been saved yet.
42     register uint32_t stackpointer __asm("r0");
43     // Context switch logic.
44     // We only save R4-R11 on the stack, because the NVIC already saved
           the other registers before calling this function.
45     __asm volatile (
46         "AND    R1, LR, #0x0D   \n\t"       // Logical AND with 0xD
47         "CMP    R1, #0x0D       \n\t"       // Use CMP to set EQ/NE flag
48         "BEQ    use_psp         \n\t"       // Branch to use_psp if EQ is
               set
49         "BNE    use_msp         \n\n"       // Branch to use_msp  if NE is
                set
50         "use_psp:               \n\t"
51         "MRS    %0, PSP         \n\t"       // Move Process Stack Pointer
               to R1 (the stackpointer variable)
52         "STMFD  %0!, {R4-R11}   \n\t"       // Store registers R4 to R11
               on the stack. (The FD in STMFB means fully descending [stack])
53         "MSR    PSP, %0         \n\t"       // Update the actual PSP
               stackpointer register with the new value after STMFD.
54         "B      exit            \n\n"       // Skip over the use_msp
               routine
55         "use_msp:               \n\t"
56         "MRS    %0, MSP         \n\t"       // Move Master Stack Pointer
               to R1 (the stackpointer variable)
57         "STMFD  %0!, {R4-R11}   \n\t"       // Store registers R4 to R11
               on the stack.
58         "MSR    MSP, %0         \n\n"       // Update the actual MSP
               stackpointer register with the new value after STMFD.
59         "exit:                  \n\t"
60         : "=r" (stackpointer)
61         : :
62     );
63     return stackpointer;
64 }
```

Listing 5.5: *src/kernel/context.h*

as that read from the LR register when the process was interrupted. Figure 5.10 shows the flow when switching contexts. Listing 5.6 shows the source code that restores the context of a process and branches to the EXC_RETURN.

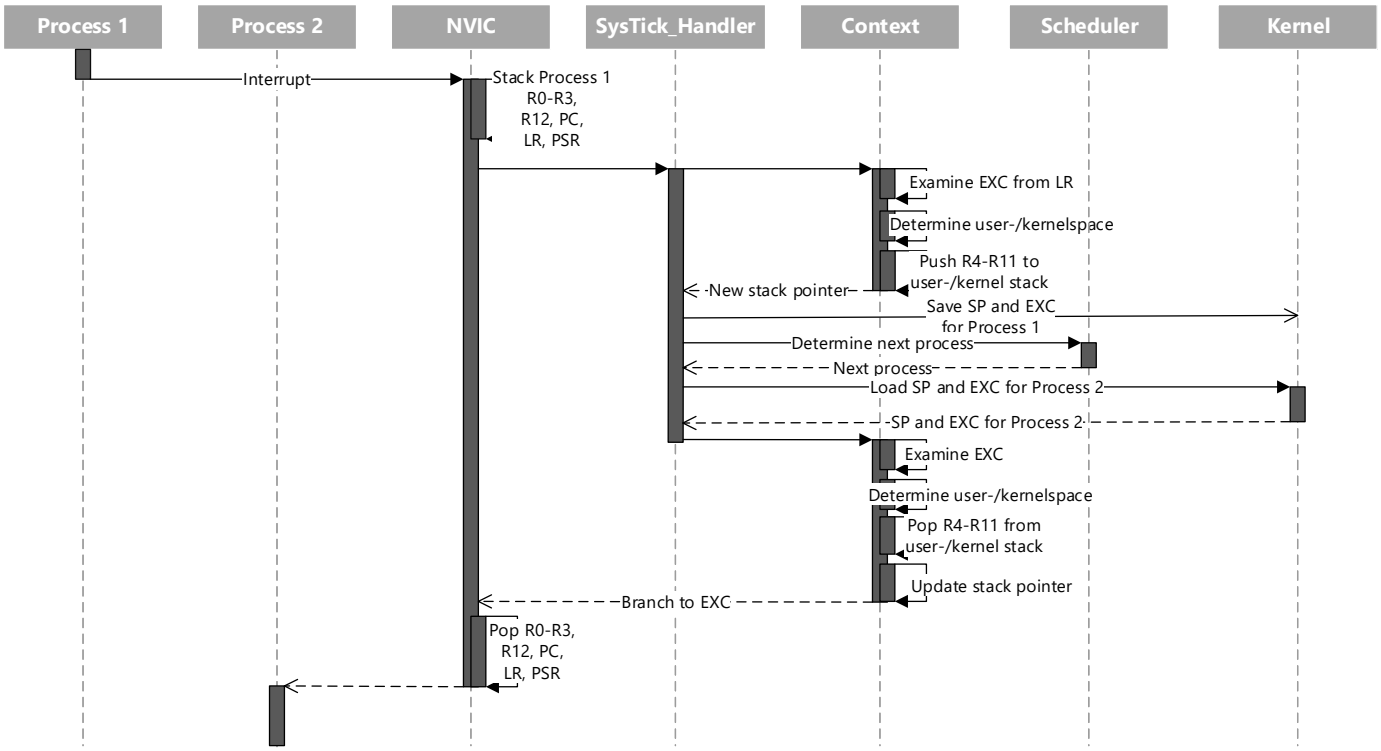


Figure 5.10: This figure shows the steps involved in switching context from one process to another

```
66 __attribute__((always_inline))
67 static inline void context_restoreAndSwitch(uint32_t stackpointer, uint8_t
       exc_return_value)
68 {
69     // Restore the software context of the new process.
70     // Variable data is loaded into static registers in the beginnning
71     // to make sure the GCC doesn't decide to use the registers R4-R11
           after
72     // they've been restored; that would be catastrophic!
73     __asm volatile (
74         "MOV     R0, %[stack]              \n\t"       // Move the stack
               pointer to R0
75         "MOV     R2, %[exc]                \n\t"       // Move the process
               specific EXC_RETURN value to R2
76         "LDR     R1, =0xFFFFFF00           \n\t"       // Load R1 with the
               constant part of EXC_RETURN values
77         "LDMFD   R0!, {R4-R11}             \n\t"       // Restore the stacked
                registers R4-R11
78         "TST     R2, #0x4                  \n\t"       // Test bit 2 of
              EXC_RETURN
79         "ITE     EQ                        \n\t"       // Which stack pointer
                was used?
80         "MSREQ   MSP, R0                   \n\n"       // Update MSP if EQ is
                set
81         "MSRNE   PSP, R0                   \n\t"       // Update PSP if NE is
                set
82         "ADD     R1, R1, R2                \n\t"       // Add the process
               specific part of the EXC_RETURN value to the constant
83         "BX      R1                        \n\t"       // Branch to the
              EXC_RETURN value to active the NVIC's context restore
84         : : [stack] "r" (stackpointer), [exc] "r" (exc_return_value) :
85     );
86 }
```

Listing 5.6: *src/kernel/context.h*

5.8.2.3 Windows

Windows are the abstract concept used by system integrators to schedule partitions in ARINC 653. A window consists of a start time, relative to the major frame, and a duration. These windows are specified by the system integrator in the XML schema (see section 4.7).
Time is allocated to partitions according to the windows specified in the XML schema, but windows do not have to be contiguous; there may be unused time between the end of one window and the start of the next. In that case, this time is considered to be an implicit idle period.

To see an example of how windows from the XML are reordered in the kernel, and how implicit idle periods are filled in, see figure 5.9.

5.8.2.4 Partition Scheduler

During the initialisation of the system, a static schedule is built based on the windows specified in the XML schema (see section 5.8.1.3).

The partition scheduler is supposed to keep track of the position within a major frame, and based on the position, figure out which window that corresponds to, and then schedule the partition of that window.

The scheduler keeps track of the position within a major frame with the HAL tick. This tick is incremented every millisecond. This is the same tick used by the delay function (section 5.4.4). The position within each major frame is derived by the tick count modulo the length of a major frame in milliseconds. If the result of this operation is zero, it is assumed that a new major frame has begun. Otherwise, if the position within the major frame is on or after, the active window closes. The scheduler then switches to the next window, and schedules the next partition. Although the time at which a window closes can be derived from the start time plus the duration, the end time of the active window is cached in memory to avoid repeated calculation of the end time.

5.8.2.5 Process scheduler

Processes are scheduled based on priority and state. Of the processes in the READY state, the one with the highest priority should be scheduled.

The process scheduler first iterates through all the processes of a partition and selects the first available process in the READY state. However, the processes are not stored in a ready queue, but are instead stored only in the order they were created. Therefore, finding the first ready processes does not guarantee that it is also the processes with the highest priority. The scheduler therefore continues to iterate through the available processes to try to find a better fitting process. For each iteration, if a process is found, it is selected as the base for further iterations. At the end of the loop, the process in the READY state with the highest priority will be selected.

If there are no processes in the READY state, none will be selected, and kernel will fault, so each partition must have at least one process in the READY state.

5.8.3 Ports

Interpartition communication (section 4.6.2) is implemented with an initialisation function and a function for every APEX call to the given port type.
As described in 4.6.2, a generic port struct as referenced in listing 5.7 is used to define a port with all its attributes. All methods operate on the data in these structs, which also contains all messages received.

```
13 struct queuing_port {
14     QUEUING_PORT_STATUS_TYPE;
15     circBuf_t                   circ_buf;
16     uint8_t                     *buffer;
17 };
18
19 struct sampling_port {
20     SAMPLING_PORT_STATUS_TYPE;
21     uint8_t                     *buffer;
22 };
23
24 typedef struct {
25     bool                        is_queuing_port;
26     union {
27         struct queuing_port     q_buf;
28         struct sampling_port    s_buf;
29     };
30     bool                        activated;
31     void                        *channel_link;
32     NAME_TYPE                   portname;
33 } port_t;
```

Listing 5.7: *src/kernel/ports.h*

Listing 5.7 shows how every `port_t` has the four following common variables:

- The first being a boolean indicating whether the port is a a queuing port or sampling port
- Another boolean named `activated`, indicates whether or not a given process has created the port. Since all attributes are statically allocated, requesting a port to be created only allows a port to transmit data, hence the attribute `activated`, is used to check whether or not a port has been created and can be used to send or receive
- A void pointer `channel_link` is used to reference the channel which the port belongs to. This pointer is said at when all the ports are initialized.
- `portname` is the name of the port, as a string.

As for the port attributes specific to the port type, they are declared as a union of the two different port attribute types `struct queuing_port` and `struct sampling _port`. Both types contain a byte array to hold buffered messages. The buffer of the sampling port is a simple buffer with the size of maximum message size, while the buffer of the queuing port is used as a circular buffer, hence the use of an object with the type `circBuf_t` to hold additional data, used when pushing and popping messages.

All code used to manipulate a circular buffer is contained src/kernel/circular_buffer.c and is not be covered in this report.

Besides arrays to contain the buffers each type of port also has a port status struct of type `QUEUING_PORT_STATUS_TYPE` and `SAMPLING_PORT_STATUS_TYPE`. These types are declared in `apex_queuing.h` and `apex_sampling.h` from the folder `src/ third_party/ARINC/APEX`. They contain the specific status attributes for the port type. The structs are embedded (listing 5.7) in as anonymous structs meaning that one can access its members directly instead of as a conventional substruct. While this is not strictly a native C feature it is enabled in GCC with the `-fms-extensions` flag and is only used to make the code easier to read and edit.

While the ports.h file is used to declare all information about the ports, the functions to work on sampling ports and queuing ports are located in src/kernel/sampling_port.c and src/kernel/queuing_ports.c respectively.

For the rest of section 5.8.3, queuing ports will be used to explain how interpartition communication is implemented in these files.

As mentioned above, ports are created by "activating" a statically allocated port. This is done from a process, by a system call. Specifically by calling the function `CREATE_QUEUING_PORT()`. When making this call, a kernel function named `create_queuing_port()` is called which takes care of the actual setup.

The same scheme of having a similar named kernel function for every system calls is used for everything which has to do with the ports.

Since the attributes and starting values for the ports is auto-generated at compile-time, the setup can be kept simple:

```
51 (void) QUEUING_DISCIPLINE;
52
53 partition_t *this_partition = getActivePartition();
54 port_t *ports = this_partition->ports;
55
56 for (APEX_INTEGER n = 0; n < this_partition->nb_ports; ++n) {
57     if (!strcmp(ports[n].portname, QUEUING_PORT_NAME) &&
58     ports[n].q_buf.MAX_MESSAGE_SIZE == MAX_MESSAGE_SIZE &&
59     ports[n].q_buf.MAX_NB_MESSAGE == MAX_NB_MESSAGE &&
60     ports[n].q_buf.PORT_DIRECTION == PORT_DIRECTION)
61     {
62         ports[n].activated = true;
63         *QUEUING_PORT_ID = n;
64         *RETURN_CODE = NO_ERROR;
65         return;
66     }
67 }
68
69 *RETURN_CODE = INVALID_PARAM;
```

Listing 5.8: *src/kernel/queuing_port.c*

Since open653 only features a simplified implementation of the ARINC 653 standard, only a minimal functioning featureset is present for the ports.
The input variable `QUEUING_DISCIPLINE` for example is in this case not used, hence it has been casted to type `void` as seen in listing 5.8 at line 51.

The remaining input variables are checked for their validity after the current port in question has been located. If these are found to be consistent with the auto-generated attributes, the port is activated by setting the `activated` attribute to true (line 62). An ID is also set as a return value and an error is cleared before returning. If no port is found to match the input attributes passed to the function the APEX defined `INVALID_PARAM` error code is passed instead (line 69).

The functions `write_sampling_message` and `read_sampling_message` follows the same pattern:
1. Validate if the input is consistent with the port information.
2. Update port status and/or pop/push message to buffer.

Only the functions `get_queuing_port_id()` and `get_queuing_port_status()` are different in that they just return information about a certain port.

The code for the sampling port, follows the same pattern and much of the code is in fact the same. Messages are passed in a different manner and circular buffers

are not used. Instead a simple buffer is overwritten with new messages or being read from, which makes it simpler to implement than queuing ports, having to use less code for managing the buffer.

5.8.4 System/APEX Calls

System calls are implemented with the ARMv7M instruction `SVC` (Supervisor Call). When this instruction is executed, a interrupt will be triggered. Already set up by the CMSIS library, any SVC interrupts will trap into the function named `SVC_Handler`. In order to avoid having to deal with assembly[9] code in partition level software, system call functions (including the APEX functions) are implemented as a library that entirely handles the system call. The sequence of steps for making a system call is a follows:

1.	Userspace	Partition code calls one of the system call wrapper functions.
2.	Userspace	The wrapper function moves the number denoting the system call function into register R0.
3.	Userspace	The wrapper function moves any input arguments into the registers R1-R3 and R12. If the function has more than four input arguments, those must be stacked.
4.	Userspace	The wrapper function issues an `SVC` instruction.
5.	Hardware	The NVIC stacks the registers R0-R3, R12, PC and LR and branches to the `SVC_Handler` function.
6.	Kernelspace	The `SVC_Handler` function investigates the stacked value of register R0 to determine which function the partition level code wished to execute.
7.	Kernelspace	The `SVC_Handler` function calls the appropriate function with the input arguments from the stacked registers.
8.	Kernelspace	The `SVC_Handler` puts the results of the function in the stacked registers. The return code is moved to R0, and additional output is put in the registers R1-R3 and R12. If necessary, more output is put in the stack.
9.	Kernelspace	The `SVC_Handler` returns.
10.	Hardware	The NVIC restores the registers R0-R3, R12, PC and LR from the stack and returns execution to the system call wrapper function.
11.	Userspace	The wrapper function returns the output from the system call as stored in the registers to the calling partition code.

Only the registers R0-R3 and R12 are available for argument passing, as these are the only general purpose registers that are stacked by the NVIC. The NVIC does

[9] It is necessary to use assembly code to issue an `SVC` instruction

Identifier	Name
0xC0FFEE0A	CREATE_PROCESS
0xC0FFEE0B	GET_TIME
0xC0FFEE0C	CREATE_QUEUING_PORT
0xC0FFEE0D	RECEIVE_QUEUING_PORT
0xC0FFEE0E	SEND_QUEUING_PORT
0xC0FFEE0F	GET_QUEUING_PORT_ID
0xC0FFEE10	GET_QUEUING_PORT_STATUS
0xC0FFEE11	PROCESS_STOP_SELF

Table 5.2: A table of the implemented system calls and their identifier.

not clear the registers before entering the `SVC_Handler` function, and thus all the general purpose registers may contain the same values as when the `SVC` instruction was executed, but their consistency is not guaranteed, as other interrupt service routines could have run before the `SVC_Handler` function started execution.
To avoid rescheduling occurring during a system call, the SysTick interrupt is turned off during the execution of the `SVC_Handler` function.
Each system call is identified by the number the caller has moved into the R0 register before issuing the `SVC` instruction. The `SVC` instruction does supply a way to embed this number directly as a part of the instruction as an immediate value, instead of using a separate register, but dedicating a register to this was chosen for two reason: One, it is poorly documented how software can read the value embedded in the instruction, and two, to keep it consistent with how system calls are handled on other architectures such as x86.
Table 5.2 shows all the implemented system calls and their identifier.

5.8.5 Error Handling

One of the features of the Health Monitor of ARINC 653 is to handle faults occurring in partitions, and do an action that has been defined in the XML schema, e.g. restart the partition. As this is not implemented, there is no defined fault handling. To keep the implementation simple, Open653 has no automatic error recovery; all errors detected by the hardware (privilege violation, memory access violation, etc.) are handled by halting the system. When the hardware throws an exception, Open653 handles these by dumping the values of the CPU registers at the time the exception was trigger over the UART connection. To make it more visible that the system has halted, all LEDs are turned on. Finally the system halts. The system can be reset by either reconnecting the power or pressing the dedicated reset button on the Mini-M4 board.

```
58 typedef struct {
59     uint8_t                  id;
60     NAME_TYPE                partitionname;
61     CRITICALITY              criticality; /* Not used */
62     bool                     systempartion; /* Not used*/
63     void                     (*entrypoint)(void);
64     APEX_INTEGER             nb_ports;
65     port_t                   *ports;
66     uint32_t                 nb_processes;
67     uint32_t                 index_running_process;
68     process_t                processes[
           MAX_PROCESSES_PER_PARTITIONS];
69 } partition_t;
```

Listing 5.9: *src/kernel/types.h*

For the drivers, if they are not initialised properly, the HAL library would notify the error handler that it encountered an error, and again, turn on the LEDs and halt the system.

5.9 Partitions and processes

Partitions and processes are implemented data structures as described in section 4.9 of System Design.

5.9.1 Partitions

Partitions are wrapped in the struct `partition_t` (see listing 5.9). The data in this data structure is filled in by both the XML schema parser, through the xml_data.c file, and by the kernel itself. The XML schema parser defines all but process information; as processes are dynamic, they are not defined in the XML schema.
As there is no compile-time knowledge of the number of processes each partition may create, the `processes` array has a static size of 3. This makes the memory consumption of a partition in the kernel more consistent, but moreover, it makes the dynamically created processes statically allocated.

Partition memory spaces are allocated in the XML schema, and they are parsed by the XML schema parser and made available to the kernel in xml_data.c. Listing 5.10 shows the memory data structures of the generated XML data.

```
71 typedef struct {
72     mem_type_t                  type;
73     uint32_t                    size;
74     mem_access_t                access;
75     uint32_t                    address;
76 } mem_req_t;
77
78 typedef struct{
79     uint8_t                     id;
80     NAME_TYPE                   partitionname;
81     uint32_t                    arr_size;
82     mem_req_t                   *memory_arr;
83
84     /* This value could be calculated every time we make a new
           process,
85     but this is just way easier... */
86     uint32_t                    mem_offset;
87 } part_mem_t;
```

Listing 5.10: *src/kernel/types.h*

If time had allowed it, the space separation of partitions would have been ensured by the MPU (see 4.6.4). Time separation of partition is handled by the partition scheduler (see 4.6.1 and 5.8.2.4).

Partitions run in unpriviledged mode; to access control registers, and to call APEX functions, they need to do a system call (see 4.6.6 and 5.8.4).

Partitions do not directly execute any code, but rely on default processes. As partitions have no directly executable code, there is nothing to go to, if no processes are ready to execute. Therefore, it is important that each partition has at least one process in the READY-state during its windows.

5.9.2 Processes

Processes are wrapped in the struct `process_t` (see listing 5.11). The `process_t` data structure covers over the ARINC 653 defined attributes, such as priority, state, stack size, etc. through an anonymous struct (PROCESS_STATUS_TYPE) defined in the appendix of the ARINC 653 specification[40]. Additionally, the data structure contains three Open653 specific attributes: the memory address of the stack-pointer (set and read only during context switching), the EXC_RETURN value generated by the NVIC when the process was last pre-empted (see table 4.2), and a timestamp denoting when the process last transitioned from the DORMANT or WAITING state to the READY state. This timestamp is set to and compared with

```
51 typedef struct {
52     uint32_t                    stackpointer;
53     uint8_t                     exc_return_value;
54     uint32_t                    tickStamp;
55     PROCESS_STATUS_TYPE; /* unamed struct with -fms-extensions */
56 } process_t;
```

Listing 5.11: *src/kernel/types.h*

the `HAL_GetTick()` function, hence the name *tick*Stamp.
Process ID is not saved as part of the process data structures, but instead denotes the index of the process in the partition data structure (see listing 5.9).

Now, processes are allocated from the pool of available processes in the `partition_t` data structure. Unallocated processes are identified by their state. If a processes is in the DORMANT state, it is considered unallocated. Process 0, the main process of a partition, is allocated during the OS initialisation, and it is the main function (in ARINC 653 terms, the entrypoint), of a partition. From userspace, processes can create other processes through the APEX call `CREATE_PROCESS` (see the ARINC 653 specification for details about this call).
There is currently no way for a processes to enter the WAITING state; if a process needs to wait, it must be implemented as a busy wait. Processes can, however, transition themselves back into the DORMANT state through the the APEX call `STOP_SELF` (see the ARINC 653 specification for details about this call).

5.9.3 idle_sys

The idle partition, technically named idle_sys, is an implicitly generated partition. This partition is used to fill out potentially empty time between different windows (see 4.6.1). This partition should not be specified in the XML schema, but will be automatically defined by the XML parser. This partition will always be the partition with index 0. The reason this partition is kept visible in the generated xml_data.c file, is to make it more transparent to the system integrator what is running during idle periods.

The purpose of this partition is to have something to switch to, when nothing else is scheduled, while using the minimum amount of resources possible. To keep the partition as lightweight as possible, the partition only does a `NOP`-loop. See listing 5.12 to see the source code of the idle partition.

```
1 idle_main:
2         NOP
3         NOP
4         NOP
5         B    idle_main
```

Listing 5.12: The entire code of the idle partition (excluding the assembly flags)

The partition code is written in assembly to make sure it uses the least possible resources. While executing, the code uses no memory, although it needs at least 32 bytes to store the hardware stacked registers during a context switch. The reason the code consists of three `NOP` instructions before a branch, is to fill the pipeline[10] with `NOP` operations before each branch. It is assumed, although no attempt has been made to prove it, that this will allow the CPU to run slightly more efficient.

5.9.4 stdio_sys

The stdio_sys partition is named after the 'standard IO' (stdio.h) from the standard C library, with the _sys being present to indicate that it is a system partition. User partitions should only interface with the rest of the system through APEX calls, so not to enter kernel space and directly communicate with the UART. This partition makes this interface possible by offering a port, to send strings or raw data over the UART.

stdio_sys' entrypoint is `stdio_sys_main()`, which only starts a new process with the `worker()` function, shown in listing 5.13.

[10]The pipeline of the Cortex-M4 consists of 3 stages

```
34 void worker(void)
35 {
36     QUEUING_PORT_ID_TYPE QUEUING_PORT_ID;
37     RETURN_CODE_TYPE RETURN_CODE;
38     CREATE_QUEUING_PORT("sys_stio", 32, 32, DESTINATION, FIFO,
39                 &QUEUING_PORT_ID, &RETURN_CODE);
40
41     char str[1024];
42     while(1) {
43         RETURN_CODE_TYPE RETURN_CODE;
44         MESSAGE_SIZE_TYPE len;
45         RECEIVE_QUEUING_MESSAGE(QUEUING_PORT_ID, 0, (uint8_t *)str,
46                         &len, &RETURN_CODE);
47         if (RETURN_CODE == NO_ERROR) {
48             BSP_UARTx_transmit((uint8_t *)str, len);
49         }
50
51         HAL_Delay(10);
52     }
53 }
```

Listing 5.13: *src/partitions/stdio_sys/stdio_sys.c*

Lines 36-39 in the listing show how the port `sys_stdio` is initialized and lines 43-49 show how messages are read from the port, and how on success the messages are transmitted on the UART with the `BSP_UARTx_transmit()` function.

5.9.5 yellow_toggler and red_toggler

Two standard partitions are implemented called `yellow_toggler` and `red_toggler` to demonstrate partition use. Their job is to send messages to the stdio_sys through queuing ports and to toggle the LEDs. Only `yellow_toggler` is explained here, since they both do the same.

Just like the `stdio_sys` partition in section 5.9.4, the only job of the function's entrypoint is to initialise a new process `worker`, which is shown in listing 5.14.

```
34 void worker(void)
35 {
36     QUEUING_PORT_ID_TYPE QUEUING_PORT_ID;
37     RETURN_CODE_TYPE RETURN_CODE;
38     CREATE_QUEUING_PORT("yellow_print", 32, 32, SOURCE, FIFO,
39                 &QUEUING_PORT_ID, &RETURN_CODE);
40
41
42     while (1) {
43         char *str = "Yellow toggler\n\r";
44         size_t len = strlen(str);
45         RETURN_CODE_TYPE RETURN_CODE;
46         SEND_QUEUING_MESSAGE(QUEUING_PORT_ID, (uint8_t *)str,
47                         len, 0, &RETURN_CODE);
48
49         for (size_t i = 0; i < 10; i++) {
50             onboard_led_toggle(yellow_led);
51             delay_ms(100);
52         }
53     }
54 }
```

Listing 5.14: *src/partitions/yellow_toggler/yellow_toggler.c*

Lines 36-39 in listing 5.14 show how the port `yellow_print` is initialised and lines 43-47 show how messages are sent to the `stdio_sys` partition. For every message the yellow led is toggled 10 times with 100 milliseconds inbetween (lines 49-52). In a complete ARINC 653 compliant operating system, direct calls to the driver to toggle LEDs should not be possible, but might be implemented with the use of sampling ports. This has not been addressed at this stage of development, because of time restrictions. The same goes for the `delay_ms()` function (line 48), used to delay a process with a time interval. This functionality is ideally accessed through an APEX call, with added integration with the process scheduler. The same is the case for the `stdio_sys` partition in section 5.9.4.

5.9.6 The evil partition

The evil partition is implemented to validate the space separation of partitions. In two ways, it attempts to break the space separation and change either memory belonging to other partitions or branch to code belonging to another partition.
By default, both of these attempts to break the space separation are disabled. This is because the MPU is not currently being used to protect memory. Additionally, as the Health Monitor is not implemented, issuing an instruction that requires privilege or modifying a register that requires privilege will crash the entire system, not only the partition. Therefore, no test is done to validate that partitions do in

fact run in unprivileged mode.

5.10 Features of the system

Presented is a short list of features implemented in Open653.

Service	Completeness
Schema parser	Parses an XML file to C structs to provide the information required by Open653
Partition management	A RT partition scheduler
Process management	A simple pre-emptive process scheduler
Memory management	MPU driver
Time management	Timer with a basetime of 5 nanoseconds and RTC for error logging
Interpartition communication	Queuing and sampling ports
Intrapartition communication	Not implemented
Health monitor	Basic error notification
Debugging	Serial interface and JTAG

CHAPTER 6

Testing

6.1 Memory Mapping Algorithm

The sizes of memory regions are powers of two. This means that the more memory a partition uses, the more memory it can potentially waste. To combat this waste, partitions can share memory regions, and be separated into subregions. This will allow the same level of safety, but more control. To help optimise the memory layout of partitions a utility program is implemented that can sort the memory regions of partitions. Each region can contain 8 subregions. All subregions within a region are of equal size.

The following is an example of how this utility program can help optimise a memory layout:

Assume five partitions with the sizes 20, 65, 120, 80 and 40 bytes, as seen in figure 6.1.

Figure 6.1: An example of 5 partitions of different sizes

Some of these partitions will waste a lot of memory space if they are allocated their own memory region. Two of the partitions in particular, the ones with the sizes 65 and 80 bytes, because they only barely cross the 64-byte barrier, needs to be allocated 128 bytes each.
Without optimising the memory layout of the partitions, the memory needed for all of these would be 480 bytes, which leaves 155 bytes unused. Figure 6.2 illustrates how each partition would fit in a memory region, and how much of that memory would be wasted.

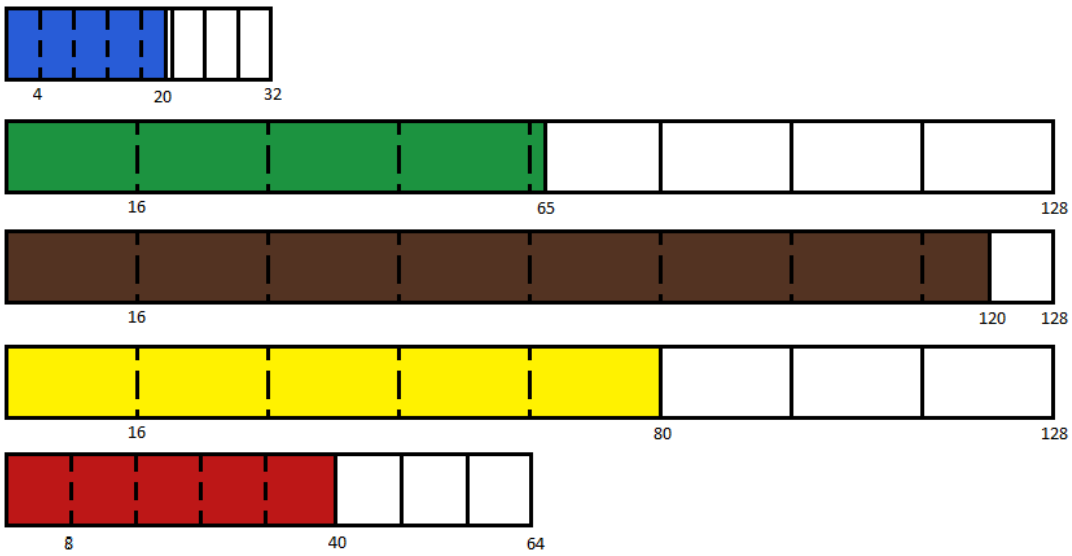


Figure 6.2: An example of a partition memory layout when no partition share a single MPU memory region

The algorithm for optimising the memory layout is simple. First, allocate a memory region to the biggest partition. If this partition leaves any unused subregions, an attempt is made to find the biggest partition fitting into those. This is continued until all partitions are allocated space. Once optimised, the five example partitions will only use 384 bytes of memory. Figure 6.3 shows the optimised memory layout.

Figure 6.3: An example of an optimised partition memory layout

Partitions are additionally padded, in case they do not fill an entire the subregion.

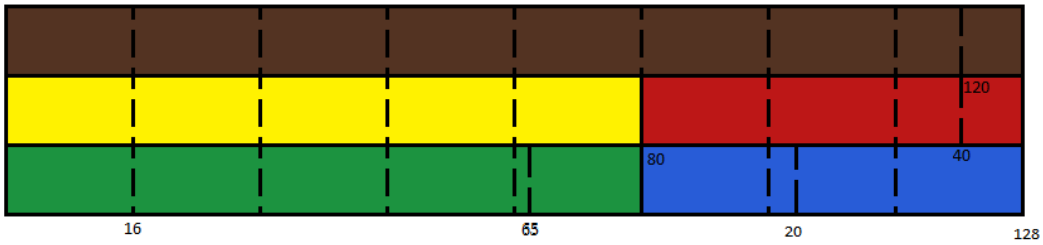


Figure 6.4: An example of how partition memory is padded to fill entire subregions

In order to test the memory partition mapping, the sorting algorithm is applied and the final memory distribution printed to a terminal. The test variables are: memory size, and partition size. When testing the partition size may change. The output in figure 6.5 shows the space given to each partition and the location in the memory map.
The results show the new region sizes containing the larger partitions and sub-regions containing the smaller partitions. The results correspond to the mapping algorithm.

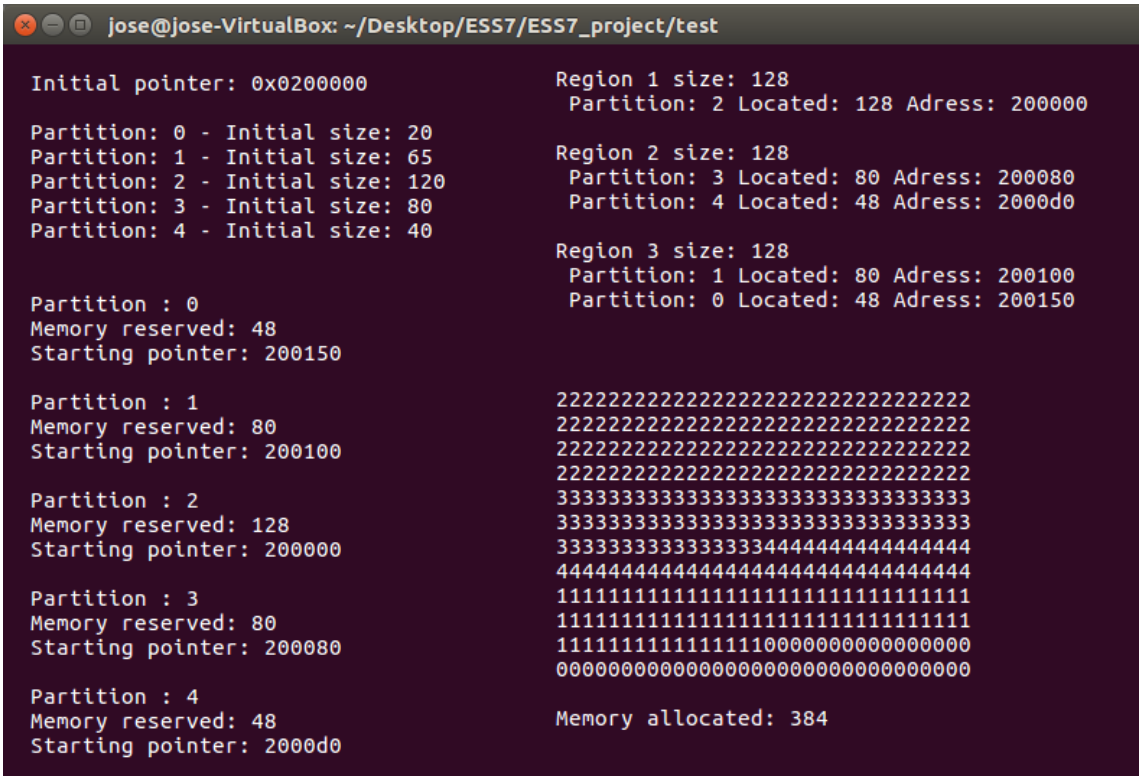


Figure 6.5: Terminal displaying the results

6.2 XML validation

Currently there is no validation of the XML file. The XML is currently prone to human error when manually adding to the file, also it does not prevent elements not compliant with the schema being written.

6.3 Scheduler

The scheduler has been tested to work provided the XML file specifies reasonable windows. The scheduler has not been tested with edge cases.
The kernel does not record any runtime metrics. Empirical evidence shows that the scheduler is working according to the windows specified in the XML schema and according to the designed algorithms. Window durations is scaled up to make it possible with the human eye to determine that the scheduler indeed follows the windows.

6.4 ARINC 653 specification - Part 3

In order to test a system for compliance with ARINC 653, the authors of the standard provide a conformity test specification. This is a separate document, as a part of the ARINC 653 specification:

- **Part 0** Introduction to ARINC 653
- **Part 1** Required services. This includes system services, data structures and functional behaviour.
- **Part 2** Extended services. For example file handling or external events
- **Part 3 Conformity test**
- **Part 4** Subset Services
- **Part 5** Core Software Recommended Capabilities

Software developers should use this to test the compliance with Part 1 of the standard. This could be done in the future phases of the Open653 OS in order to demonstrate full compliance of the APEX behavior.

6.5 Interpartition Communication

Interpartition Communication is tested for queuing ports, not sampling ports. The test is done with the three partitions `red_toggler`, `yellow_toggler` and `stdio_sys`. `red_toggler` and `yellow_toggler` broadcast their respective partition name as a string to the `stio_channel`. The `stdio_sys` reads any message within its timeframe and transmits the strings by UART to a connected terminal. Both transmitting partitions are set to transmit their string every two seconds. A sample of a recorded output is depicted in figure 6.6, showing that the communication scheme works as intended.

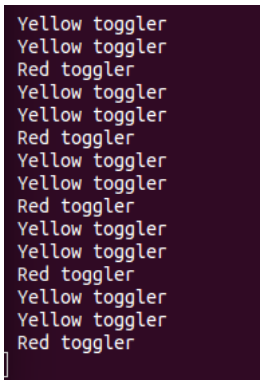


Figure 6.6: Terminal showing the strings originating from two different partitions.

As can be seen from the figure, the `yellow_toggler` is sending twice as many messages as `red_toggler`. This is due to the scheduling of the two partitions. The file main_schema.xml shows that both partitions are scheduled to execute twice within the major time-frame of 20 seconds, but that `yellow_toggler` is given twice the time of `red_toggler` at every interval and hence gets to print more often.

Even though the sampling port module is implemented in the codebase, it is yet to be fully integrated into Open653 and therefore is not tested.

CHAPTER 7

Conclusion

Open653 is implemented and capable of parsing and compiling the XML, to setup and run partitions, providing a working product that can be observed physically in the real world. The scope defined and implemented demonstrates the execution of space and time partitioned applications.
Despite this success, many of the features are not complete or contain compliance mistakes. Small inconsistencies exist due to lack of foresight as the project evolved and members collaborated in the same domains of the system.

Ultimately the goal has been achieved and the foundation for future development has been set. The questions presented in the Problem statement can be addressed as follows:

Which features must be developed to make Open653 functional and testable?
The features required to make the system functional and testable are listed in Essential components section 4.1.1 and the implemented features are listed in section 5.10.

How can this system be developed without the use of pre-existing implementations?
Open653 is developed as a ground up implementation on top of the MINI-M4. The HAL library has been used to supplement the development of most drivers and hardware specific functions.

How is this system implemented using C and Python?
Python is used for parsing the XML into C structures which are used for building the source code. C is used to program Open653 and its partitions.

What are the main obstacles when developing such a system?
The biggest challenges faced in the project were:

- Understanding the need for using special GCC attributes when constructing special functions for context switching
- Establishing a JTAG connection when assembling the hardware
- Deciding on an auto-generated data structure

As so, the Conclusion answers the main question of the project:
Can an OS implementation based on a subset of the ARINC 653 standard be developed on a Mini-M4 for STM32?

7.1 Discussion

A the project progressed certain problems stand out from the rest, as highlighted in the conclusion. The group recommends following this advice, when conducting a similar project:

- When direct manipulation of the control flow is required, GCC must be instructed *not* to auto-generate instructions
- Use the provided instructions on how to setup the hardware (Appendix B)
- It is very important to address the schema early on, as it will shape the whole project. Furthermore special care should be taken to fully understand the ARINC 653 standard

If the project were to be continued, in depth focus should be put on individual modules, to refactor and rebuild. By doing this full compliance with ARINC 653 could be achieved. A logical next step for such an implementation, would be to apply a complete suite of compliance tests.

7.2 Reflection

The implementation of Open653 as a whole, provided sufficient challenges for each of the groups members to develop themselves and follow the education's requirements.

The software development process flowed naturally. The nature of the project allowed the team to cover different areas at the same time, whilst progressing without a strict plan. The project was pursued by applying the principles of agile software development. Being the first time the group has implemented an OS from scratch, a different approach would be taken given a similar problem in the future.

Bibliography

[1] *AAURacing*. URL: http://aauracing.dk/.

[2] SYSGO AG. *PikeOS Hypervisor*. URL: https://www.sysgo.com/products/pikeos-hypervisor/why-pikeos/#tab_c87.

[3] ARM. *Application Note 179 - Bit-banding*. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0179b/CHDJHIDF.htm.

[4] ARM. *ARM® Cortex®-M4 Processor - Technical Reference Manual*. URL: http://infocenter.arm.com/help/topic/com.arm.doc.100166_0001_00_en/arm_cortexm4_processor_trm_100166_0001_00_en.pdf.

[5] ARM. *ARM Generic Interrupt Controller*. URL: http://www.cl.cam.ac.uk/research/srg/han/ACS-P35/zynq/arm_gic_architecture_specification.pdf.

[6] ARM. *ARMv7-M Architecture Reference Manual*. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0403e.b/index.html.

[7] ARM. *CMSIS core documentation*. URL: http://arm-software.github.io/CMSIS_5/Core/html/index.html.

[8] Mason Cheung. *Microsoft empowers new development opportunities in mixed reality, gaming and cellular PCs*. Dec. 2016. URL: https://news.microsoft.com/2016/12/07/microsoft-empowers-new-development-opportunities-in-mixed-reality-gaming-and-cellular-pcs/#sm.00000u22mjr5svewzvjspco7iwi64.

[9] *CMake*. URL: https://cmake.org/.

[10] Philippa Conmy. *What is Integrated Modular Avionics (IMA)?* URL: https://www-users.cs.york.ac.uk/~philippa/IMA.html.

[11] Raspberry Pi Foundation. *Raspberry Pi 3 Model B*. Feb. 2016. URL: https://www.raspberrypi.org/products/raspberry-pi-3-model-b.

[12] *GDB, GNU Project debugger*. URL: http://www.gnu.org/software/gdb/.

[13] Matthias Gerlach, Robert Hilbrich, and Stephan Weißleder. "Can Cars Fly? From Avionics to Automotive: Comparability of Domain Specific Safety Standards". In: 2011. URL: http://publica.fraunhofer.de/eprints/urn_nbn_de_0011-n-2485950.pdf.

[14] GNU. *GCC ARM Options*. URL: https://gcc.gnu.org/onlinedocs/gcc/ARM-Options.html.

[15] GNU. *GCC C Dialects*. URL: https://gcc.gnu.org/onlinedocs/gcc/C-Dialect-Options.html.

[16] GNU. *GCC Debugging Options*. URL: https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html.

[17] GNU. *GCC Linking Options*. URL: https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html.

[18] GNU. *GNU Make*. URL: https://www.gnu.org/software/make/.

[19] AERONAUTICAL RADIO INC. "ARINC Specification 653P1-2". In: (Mar. 2006).

[20] AERONAUTICAL RADIO INC. "ARINC Specification 653P1-2". In: Mar. 2006, p. 2.

[21] AERONAUTICAL RADIO INC. "ARINC Specification 653P1-2". In: Mar. 2006, p. 11.

[22] AERONAUTICAL RADIO INC. "ARINC Specification 653P1-2". In: Mar. 2006, p. 12.

[23] AERONAUTICAL RADIO INC. "ARINC Specification 653P1-2". In: Mar. 2006, p. 13.

[24] AERONAUTICAL RADIO INC. "ARINC Specification 653P1-2". In: Mar. 2006, p. 14.

[25] AERONAUTICAL RADIO INC. "ARINC Specification 653P1-2". In: Mar. 2006, p. 19.

[26] AERONAUTICAL RADIO INC. "ARINC Specification 653P1-2". In: Mar. 2006, p. 20.

[27] AERONAUTICAL RADIO INC. "ARINC Specification 653P1-2". In: Mar. 2006, p. 25.

[28] AERONAUTICAL RADIO INC. "ARINC Specification 653P1-2". In: Mar. 2006, p. 27.

[29] AERONAUTICAL RADIO INC. "ARINC Specification 653P1-2". In: Mar. 2006, p. 36.

[30] AERONAUTICAL RADIO INC. "ARINC Specification 653P1-2". In: Mar. 2006, p. 40.

[31] AERONAUTICAL RADIO INC. “ARINC Specification 653P1-2”. In: Mar. 2006, p. 42.

[32] AERONAUTICAL RADIO INC. “ARINC Specification 653P1-2”. In: Mar. 2006, pp. 14–15.

[33] AERONAUTICAL RADIO INC. “ARINC Specification 653P1-2”. In: Mar. 2006, p. 25.

[34] AERONAUTICAL RADIO INC. “ARINC Specification 653P1-2”. In: Mar. 2006, p. 28.

[35] AERONAUTICAL RADIO INC. “ARINC Specification 653P1-2”. In: Mar. 2006, pp. 65–74.

[36] AERONAUTICAL RADIO INC. “ARINC Specification 653P1-2”. In: Mar. 2006, p. 18.

[37] AERONAUTICAL RADIO INC. “ARINC Specification 653P1-2”. In: Mar. 2006, p. 45.

[38] AERONAUTICAL RADIO INC. “ARINC Specification 653P1-2”. In: Mar. 2006, p. 61.

[39] AERONAUTICAL RADIO INC. “ARINC Specification 653P1-2”. In: Mar. 2006, p. 26.

[40] AERONAUTICAL RADIO INC. “ARINC Specification 653P1-2”. In: Mar. 2006, pp. 165–167.

[41] *Joint Test Action Group*. URL: https://en.wikipedia.org/wiki/JTAG.

[42] Future Technology Devices International Limited. *TTL to USB Serial Converter Range of Cables - Datasheet*. URL: http://www.ftdichip.com/Support/Documents/DataSheets/Cables/DS_TTL-232R_CABLES.pdf.

[43] DornerWorks Ltd. *ARLX: ARINC 653 Real-time Linux on Xen*. URL: http://dornerworks.com/portfolio/arlx-arinc-653-real-time-linux-xen.

[44] Inc. Lynx Software Technologies. *LynxOS-178 RTOS*. URL: http://www.lynx.com/products/real-time-operating-systems/lynxos-178-rtos-for-do-178b-software-certification/.

[45] Trevor Martin. *CMSIS functionality*. URL: http://www.embedded.com/design/programming-languages-and-tools/4438667/Basics-of-the-Cortex-MCU-Software-Interface-Standard--Part-1---CMSIS-Specification-.

[46] MikroElektronika. *Introduction to MINI-M4 for STM32*. URL: http://download.mikroe.com/documents/starter-boards/mini/stm32/f4/mini-m4-stm32-manual-v100.pdf.

[47] MikroElektronika. *MINI-M4 for STM32*. URL: http://www.mikroe.com/mini/stm32.

[48] MikroElektronika. *MINI-M4 for Tiva™ C Series*. URL: http://www.mikroe.com/mini/tiva.

[49] libopencm3 project. *Run From RAM*. URL: http://libopencm3.org/wiki/Run_From_RAM.

[50] James W. Ramsey. *Integrated Modular Avionics: Less is More*. Feb. 2007. URL: http://www.aviationtoday.com/av/commercial/Integrated-Modular-Avionics-Less-is-More_8420.html.

[51] Dominic Rath. *OpenOCD*. URL: http://openocd.org/.

[52] Wind River. *VxWORKS 653 3.0 MULTI-CORE EDITION*. URL: http://www.windriver.com/products/product-overviews/vxworks-653-product-overview-multi-core/.

[53] Sławomir Samolej. *ARINC Specification 653 Based Real-Time Software Engineering*. Tech. rep. Faculty of Electrical and Computer Engineering, Rzeszow University of Technology, 2009. URL: http://www.scarlettproject.eu/publications/technical/ABS653RTSE_SS1-2011.pdf.

[54] Keil software. *CMSIS integration diagram*. URL: http://www.keil.com/pack/doc/CMSIS/General/html/CMSISv4_small.png.

[55] STMicroelectronics. *AN4838 Application note*. Tech. rep., p. 6. URL: http://www.st.com/content/ccc/resource/technical/document/application_note/group0/bc/2d/f7/bd/fb/3f/48/47/DM00272912/files/DM00272912.pdf/jcr:content/translations/en.DM00272912.pdf.

[56] STMicroelectronics. *RM0090 Reference manual*. URL: http://www.st.com/content/ccc/resource/technical/document/reference_manual/3d/6d/5a/66/b4/99/40/d4/DM00031020.pdf/files/DM00031020.pdf/jcr:content/translations/en.DM00031020.pdf.

[57] STMicroelectronics. "RM0090 Reference manual". In: p. 216. URL: http://www.st.com/content/ccc/resource/technical/document/reference_manual/3d/6d/5a/66/b4/99/40/d4/DM00031020.pdf/files/DM00031020.pdf/jcr:content/translations/en.DM00031020.pdf.

[58] STMicroelectronics. "RM0090 Reference manual". In: p. 716. URL: http://www.st.com/content/ccc/resource/technical/document/reference_manual/3d/6d/5a/66/b4/99/40/d4/DM00031020.pdf/files/DM00031020.pdf/jcr:content/translations/en.DM00031020.pdf.

[59] STMicroelectronics. *ST-LINK/V2 in-circuit debugger/programmer for STM8 and STM32*. URL: http://www.st.com/content/ccc/resource/technical/document/data_brief/80/76/2b/dc/45/c5/46/90/DM00027105.pdf/files/DM00027105.pdf/jcr:content/translations/en.DM00027105.pdf.

[60] STMicroelectronics. *STM32CubeF4*. URL: http://www.st.com/content/st_com/en/products/embedded-software/mcus-embedded-software/stm32-embedded-software/stm32cube-embedded-software/stm32cubef4.html.

[61] STMicroelectronics. *STM32F415xx/STM32F417xx Datasheet*. URL: http://www.st.com/content/ccc/resource/technical/document/datasheet/98/9f/89/73/01/b1/48/98/DM00035129.pdf/files/DM00035129.pdf/jcr:content/translations/en.DM00035129.pdf.

[62] STMicroelectronics. "STM32F415xx/STM32F417xx Datasheet". In: p. 74. URL: http://www.st.com/content/ccc/resource/technical/document/datasheet/98/9f/89/73/01/b1/48/98/DM00035129.pdf/files/DM00035129.pdf/jcr:content/translations/en.DM00035129.pdf.

[63] Bjarne Stroustrup. *FAG*. URL: http://www.stroustrup.com/bs_faq.html.

[64] Imagination Technologies. *Creator Ci40 IoT Hub*. URL: http://creatordev.io/ci40-iot-hub.

[65] Ubuntu. *gcc-arm-none-eabi*. URL: http://packages.ubuntu.com/trusty/devel/gcc-arm-none-eabi.

[66] Wikipedia. *DO-178B*. URL: https://en.wikipedia.org/wiki/DO-178B.

[67] Wikipedia. *DO-178C*. URL: https://en.wikipedia.org/wiki/DO-178C.

[68] Wikipedia. *POSIX*. URL: https://en.wikipedia.org/wiki/POSIX.

[69] *XML to dict*. URL: https://github.com/martinblech/xmltodict.

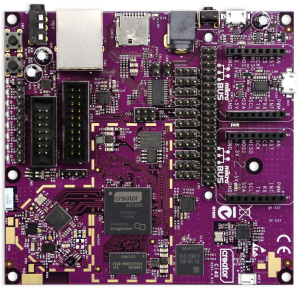
APPENDIX A

Proposed Embedded Devices

All prices listed here are given in Danish Krones (DKK), as found in the links to the shop. The links and prices are as found on the 16th September 2016

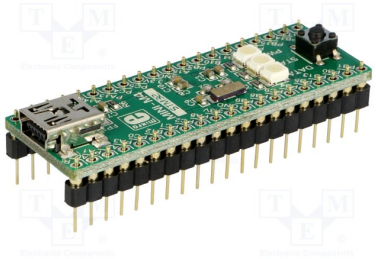
Creator Ci40

Chip manufacturer:	Imagination Technologies
Chip:	Creator cXT200
Processor:	550 MHz dual-core Ingenic JZ4780
Arch.:	MIPS
Memory	256 MB DDR3 SDRAM
Storage	2 MB Boot NOR Flash, 512 MB NAND Flash, microSD
Price:	584,60 DKK
Link to shop:	mouser.dk
Link to site:	creatordev.io/ci40-iot-hub
link to datasheet:	mipscreator.imgtec.com/.../JZ4780_PM.pdf



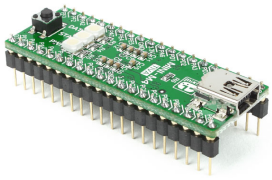
MINI-M4 for STM32

Chip manufacturer:	ST
Chip:	STM32F415RG
Processor:	168 MHz, 32-bit Cortex-M4
Arch.:	ARM
Memory	192+4 KB SRAM
Storage	1 MB Flash
Price:	206,45 DKK
Link to shop:	dk.rs-online.com
Link to site:	mikroe.com
link to datasheet:	st.com/.../en.DM00031020.pdf



MINI-M4 for Tiva

Chip manufacturer:	TI
Chip:	TM4C123GH6PM
Processor:	80 MHz, 32-bit Cortex-M4
Arch.:	ARM
Memory	32 KB SRAM, 2 KB EEPROM
Storage	256 KB Flash
Price:	212,87 DKK
Link to shop:	dk.rs-online.com
Link to site:	ti.com
link to datasheet:	ti.com/.../tm4c123gh6pm.pdf



Raspberry Pi 3

Chip manufacturer:	Broadcom
Chip:	Broadcom BCM2837
Processor:	1.2GHz 64-bit quad-core ARMv8 Cortex-A53
Arch.:	ARM
Memory	1GB LPDDR2
Storage	microSD
Price:	649,00 DKK
Link to shop:	raspberrypi.dk
Link to site:	raspberrypi.org
link to datasheet:	None exist. Use reference guide for ARM processor at: infocenter.arm.com/.../DDI0500G_cortex_a53_trm.pdf

APPENDIX B

Hardware setup

This appendix presents the step-by-step process of preparing the hardware for the software development process.
In figure B.1, the wires are setup on the breadboard, according to the pinout of the Mini-M4 board, the JTAG connector and the serial cable. Power and common ground are established on the breadboard's power rails, from the Mini-M4.

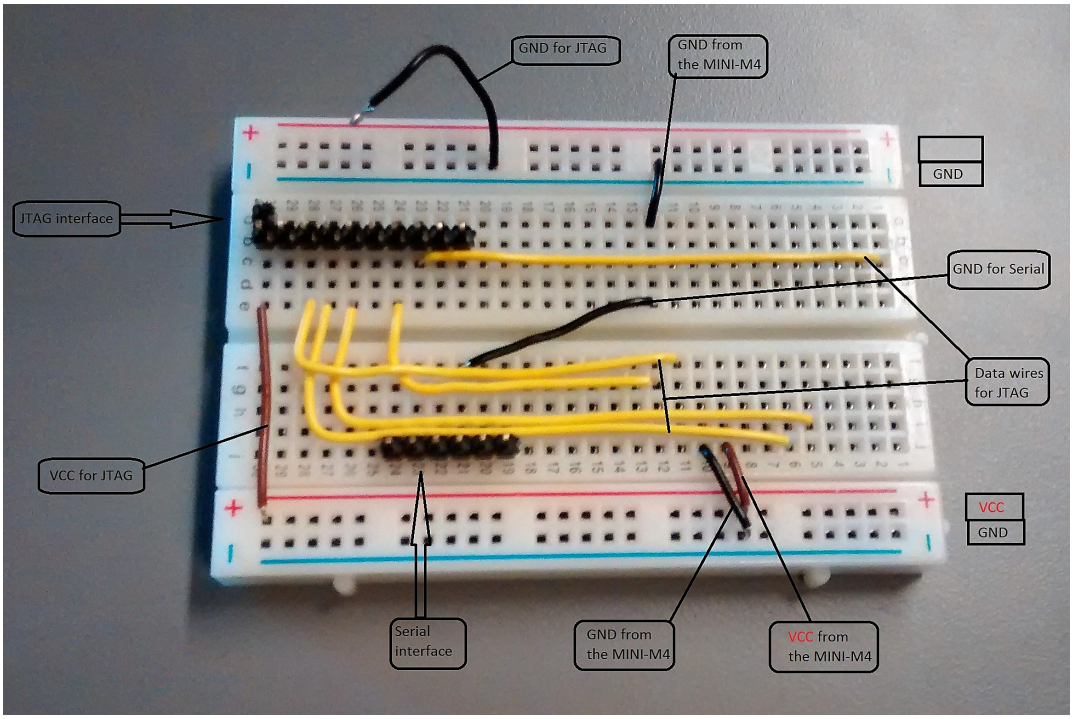


Figure B.1: The wiring on the breadboard

Figure B.2 shows the pinout of the JTAG connector. Out of the 20 pins, only five will be used by the interface, and two for getting a voltage reference into the module.

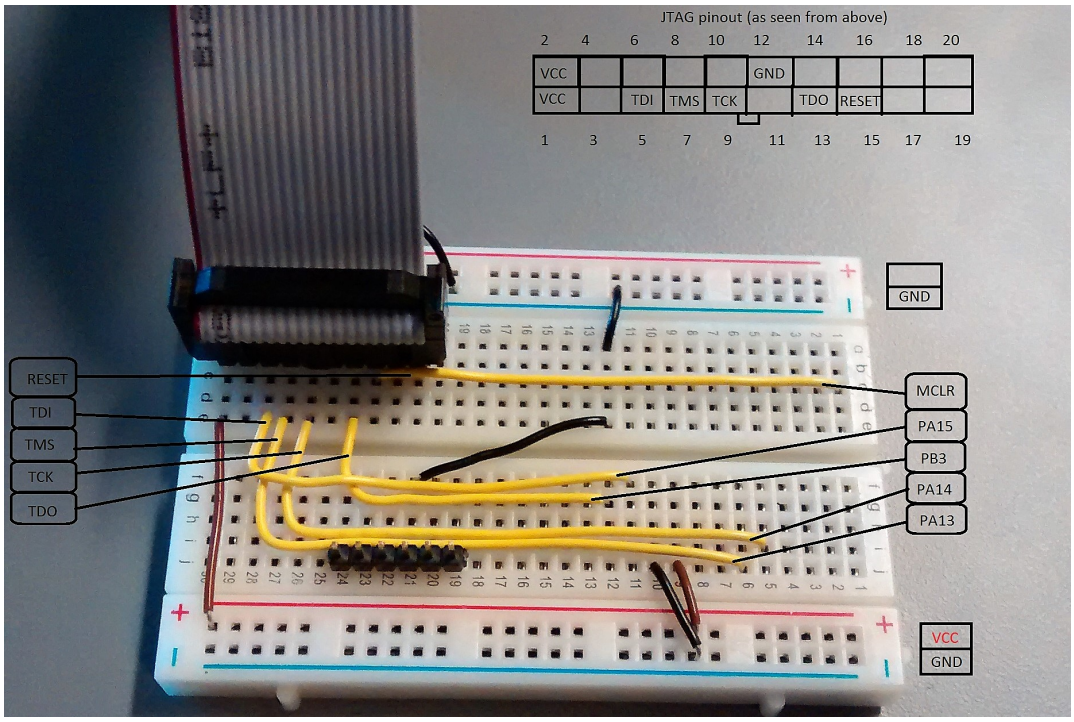


Figure B.2: Breakout of the JTAG connector

Figure B.3 shows the corresponding pins for the JTAG interface, on the Mini-M4 board. Some of the wires are hidden under the board.

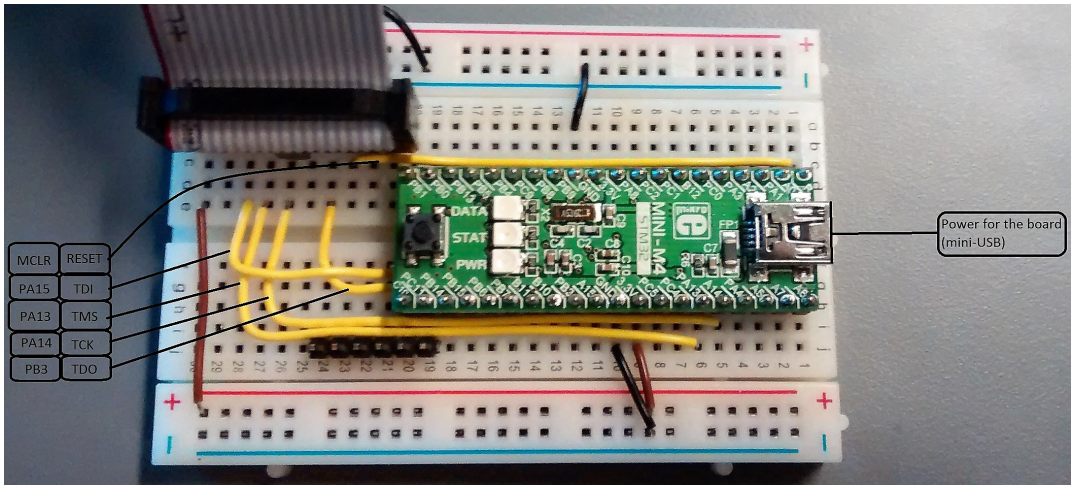


Figure B.3: The five lines of the JTAG interface and their corresponding pins on the board

Figure B.4 shows the way the serial communication was connected. The only thing to remember here is that the Tx of a device goes to the Rx of the other device it needs to communicate to.

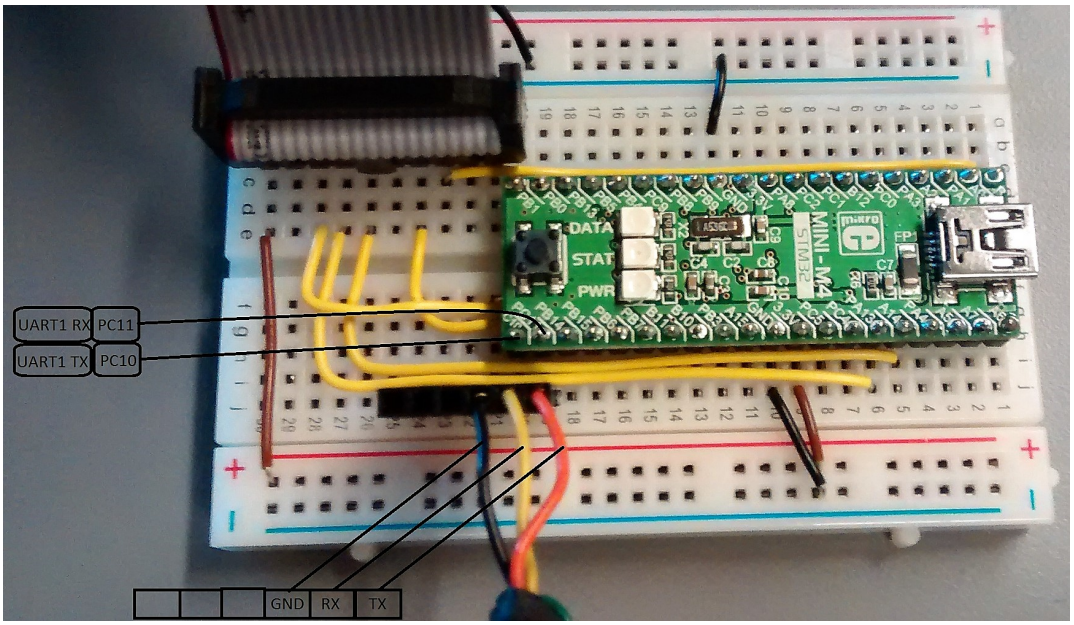


Figure B.4: Breakout of the serial communication

APPENDIX C

Compile Tutorial

This tutorial is gives a quick step-by-step introduction on how to compile the open653. It assumes that the reader already have the code and is pursuing the task using a Linux environment with the following tools already installed:

- arm-none-eabi-gcc
- make
- cmake
- openocd

The steps to compiling the OS with partitions and flashing it to the chip are as follows:

1. Locate the root folder of the project
2. Run the command "python createCStructs.py src/" from terminal
3. Make a new folder to contain the build files (eg. mkdir build)
4. From within the build fonder run the command 'cmake ..'. This reads the CMakeLists.txt files and builds make files
5. Run 'make' to build the project or use 'make help' to get a list of available targets
6. If hardware is connected the computer the command 'make OS_writeflash' can be used to compile the code and flash the hardware.

The hardware is reset after being (re-)programmed.

APPENDIX D

Forum Responses

Conversation taken from the "Embedded Related" forum in the thread [1] created by the group, to see how an open source implementation of the ARINC 653 standard is seen among professionals.

ARINC653 OpenSource Suggestions

"Hi everyone,

Me and my group are developing the ARINC653 protocol this semester in OpenSource.
The idea is to have independent partitions than can run OS, applications or other things, in well defined and restricted amounts of memory, completely independent of each others in the same platform. If one crashes it won't affect the other partitions. This was first created 10 years ago for airplanes, we are trying to adapt this to other industries. We developed the OS and the memory and time partitions already, now we are looking for different points of view about the subject, what is worth it or not to explore about the different possibilities in the future using this."

gillhern321:

"Jose, could you explain the major difference between ARINC653 and a fault tolerant OS distributed across multiple blades on a blade server configuration? The fault tolerant, fault redundant servers actually have two OS's running at the same time with multi-processing being done across 10-20 blades. So when one OS fails the failover defaults to the secondary processor and within 32ms the processing

[1] https://www.embeddedrelated.com/thread/1326/

continues same stack same interrupts. We were doing this 23 years ago at Bell Labs with the 5ESS and AMPS/PCS switching platforms, they are still in use today across all of north america they are still the core of our communications infrastructure.

Your question is, is it worth it. When you look at the processing power today versus then, and I think this is something more then worth it to all of us doing coding or developing hardware. And the potential impact to distributed processing in our home is a powerful thing to have available. OPENSOURCE OHHHH YAAAAAA. LOL"

Reply:

"Hey! Not sure if is that. We only have one main OS, that runs different partitions, which can contain independent processes, multi threads or single threads that have fixed amount of memory and processor available. If one crashes doesn't affect the others, it can shut it down or restart (up to the programmer), but all the rest keeps running normally. When the context switching happens the whole environment is "saved" and return to the same stage when the processor is given to that partition again."
Tim Wescott:

"Having worked in industry designing software as well as doing systems architecture, I'm very cynical about anyone having an up-front commitment to software safety.

Having said that, I could see such a thing being an asset in any sort of system where I wanted to partition a design by criticality levels, but keep it all on the same processor. I've always done that by putting the critical stuff on separate processors, possibly on the end of a long wire from the box with the bells and whistles – but that was in a system that naturally called for that partition anyway. If you were just bound and determined to put everything on to one processor, and you had a mix of jobs between things that would cost 1M Dollars if the software failed and things that you wanted to let sales engineers play with, then I could see value in that."

Tim Wescott:

"Doesn't this demand a processor with an MMU, and either put restrictions on who gets to write ISRs, or put demands on the processor for letting ISRs be inter-

ruptable when a new time slot comes up?"

Reply:

"We are using a MPU with 8 well defined areas for each partition.

Yes, everytime the processor changes the context, the interrupts will change to the partition in case. It has to due with the priorities.
The whole context switch completely when we go to a new partition or back to the kernel."

Tim Wescott:

"So, if I have an interrupt that's attached to some piece of hardware that only matters to my most critical task, then the most dumb-ass task might service the interrupt?
That doesn't seem right.

I would think that you'd either want a separate interrupt-handling context (operating at the highest level of safety), or that you'd want to disable any interrupts that aren't "owned" by the current context. I see problems with both of these – what does the ARINC standard say you should do?"

Tim Wescott:
"Another thought:

Anyone who wants to use this is going to be interested in safety. They may well want the software quality to be traceable back to some standard, like DO-178, or whatever that IEC standard is for medical software. Developing software to meet these standards is far more work than just whipping out any old thing – the estimate that I was given, a long long time ago (with head-nodding from everyone who had been there) is that each time you go up a level of criticality under DO-178, the software development gets 7 times more costly. Level E is "software quality doesn't matter"; level A is "smoking hole in the ground, with bodies and TV reporters". Designing software and certifying it for level A costs about 2500 times as much as doing so for level E.

So, in your market research, you may want to ask if anyone is going to buy it if it's just the typical "thrown together" quality of most desktop and phone apps."