Master's Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Measurement

Migration of an IoT Real-Time System and Reimplementation of Some of Its Functions

Bc. Jan Mrázek

Supervisor: Ing. Milan Kolář
Field of study: Computer Engineering
May 2019

Acknowledgements

I would like to thank the supervisor of this master's thesis, Ing. Milan Kolář, for providing me with a friendly working environment as well as great guidance throughout my work on this project. I would also like to thank doc. Ing. Jiří Novák, Ph.D. for giving me additional guidance and insights when authoring this text.

Furthermore I want to thank my family, who supported me not only during my work on this thesis, but also throughout my studies.

Declaration

I declare that the presented work was developed independently and I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, May 20, 2019

Abstract

This thesis is dedicated to the research and realization of a prototype for migration of a home automation system from an older, MCU-based design to a more modern solution. The older design is a microcontroller-based, low system resource solution incorporating an ARM Cortex-M4 MCU, the FreeRTOS operating system and various memory-conserving implementation approaches. The new proposed solution includes a Texas Instruments Sitara AM3358 chip and a distribution of Debian Linux patched with the PREEMPT_RT real-time kernel patch. Development is done using a BeagleBone Black single-board computer.

This thesis provides descriptions and explanations of the concepts and inner workings of real-time operating systems and the Linux device tree system. The process of reimplementing and porting of functionalities from the previous project is described, issues are discussed and their solutions explained. Performance gains are measured and explained for some of the newly implemented functionality.

Keywords: home automation, real-time operating system, Linux, device tree, PREEMPT_RT, Sitara

Supervisor: Ing. Milan Kolář

Abstrakt

Tato diplomová práce se zabývá rešerší a realizací prototypu pro migraci systému inteligentních domů ze stárnoucí koncepce založené na mikrokontroléru na koncepci modernější. Starší koncepce zahrnuje mikrokontrolér založený na jádře ARM Cortex-M4, operační systém FreeRTOS, relativně nízké systémové prostředky a několik implementačních postupů pro jejich šetření. Navrhované řešení používá čip Texas Instruments Sitara AM3358 a Linuxovou distribuci Debian včetně modifikace jádra PREEMPT_RT. Vývoj je prováděn za pomoci jednodeskového počítače BeagleBone Black.

Tato práce čtenáři poskytne popis a vysvětlení konceptů a vnitřních procesů operačních systémů reálného času a Linuxového systému device tree. V práci je popsán proces portování a reimplementace několika různých funkcí z původního systému. Problémy zjištěné během tohoto procesu jsou vysvětleny a jsou zde probrána jejich řešení. U některých reimplementací dochází k razantnímu nárůstu výkonnosti oproti předchozímu řešení - tyto změny jsou naměřeny a jejich dopad je detailněji popsán.

Klíčová slova: inteligentní dům, operační systém reálného času, Linux, device tree, PREEMPT_RT, Sitara

Překlad názvu: Migrace a reimplementace funkcí IoT systému reálného času

Contents

Figures

Tables

Chapter 1

Introduction

The market for home automation solutions has been getting bigger and bigger in recent years. To be successful in this specific field of Internet of Things and beat the vast amount of competition, home automation companies strive to create and provide solutions that are very economically (i.e. both in initial price and maintenance costs) friendly and, maybe most importantly, solutions that are intuitive to use and provide a comfortable user experience.

In order to create systems that are inconspicuous and non-invasive (to not disrupt the customer's interior design, for example), home automation developers usually design their electronics to be compact in size. This brings along the downside that a small, microcontroller-based system might not be able to provide the real estate and computing power needed to satisfy all requirements in a timely and responsive manner. When an issue of this nature presents itself to such a developer or company, there are generally multiple ways to get past it. A cost-effective way is to just limit and slow down the responsiveness of the system. While cheap, this approach is far from being correct. They might also try to pursue means of hardware upgrades to take their system back up to par speed-wise.

In some cases, home automation companies will choose to research the options and approaches to hardware upgrades even before the risks of slowdowns and issues tied to them present themselves.

I was tasked with researching such a hardware upgrade by Prague-based smart home company Prologue, s.r.o. Prologue develops and maintains a home automation system called iQtec. The plan was to research the means

of replacing the iQtec home automation system's main processing unit's CPU with a different, much more modern and faster chip. This project also included researching a switch to a different operating system - a migration from FreeRTOS to a more conventional Linux-based distribution. Another task was the migration and porting of the processing unit's codebase and functionality from the old operating system to the new one.

This master's thesis will describe to the reader in detail the process of designing and executing such a research project. The thesis consists of multiple parts; first, an overview of the the historical, current and future trends in this field of computing are discussed. Next, the existing home automation system is described. After that, some of the current system's weak points are discussed and their solutions are proposed. Then, both the existing and new systems' hardware specifics are introduced and compared. The choice of operating system is explained and some key concepts are introduced. Implementation specifics are described in detail and any issues presenting themselves during the implementation process are discussed and their solutions explained.

1.1 History and Current Trends

In the past, home automation and maybe even IoT systems as a whole used to be mostly based around simple microcontrollers and MCUs (microcontroller units). This situation rose from the fact that these generally simpler processors were typically quite cost-effective to purchase.

Low costs, however, came with a significant penalty; as a result of the relatively low RAM and storage capacity of these devices, a significant portion of the development had to be devoted to making very memory- and CPU-efficient source code. This was often extremely difficult in newly-started companies, as there simply wasn't enough manpower to be dedicated to creating a working product in a reasonable time frame.

1.1.1 External and Paid Software Toolsets

An option in such a situation was to purchase a working pre-implemented source code library or toolset from providers selling such services. There was a clear advantage in doing so, as it enabled developers to divert their

attention to implementing actual functionality. A downside of this approach was the fact that a codebase acquired in this manner often came without any sort of guarantee and was not free of bugs.

Even though these solutions were often far from perfect, their monetary costs were not insignificant, especially for a start-up operation.

There however existed paid toolsets which offered such guarantees, but those often had very different target customers - the aerospace industry. This was reflected in the price tag of such solutions, which was so extreme that no startup would elect to use them.

1.1.2 Current Market Situation

The situation in the IoT and microprocessor market nowadays contains three types of big players. First, there are companies who actually manufacture the silicon chips that are used in this type of computing. In today's tech climate, where costs are being pushed down rapidly and the period of moral obsolescence is getting relatively longer (in other words, customers don't seek hardware upgrades as frequently as before), it is very important to offer good customer support in order to sell units. Chip producers therefore put big amounts of effort into providing good documentation, reference and usage manuals and driver support.

Another sort of tech companies who represent a significant portion of the market are businesses that produce software libraries based on those drivers. There is a good amount of collaboration between these two types of tech companies - silicon manufacturers want their hardware to have a good software applicability and a good user experience for developers using it. The result of this that customers will be more likely to choose a product with a good software toolkit, thus bringing more sales to both the hardware and software producers.

These software-creating companies are also getting more and more important on the market as the complexity of the underlying hardware rises. An example - in the past, a typical microcontroller's reference manual would be hundreds of pages long. Nowadays, such reference manuals for bigger, multi-core and high feature chips are often an order of magnitude longer. The amount of information needed to get through when programming drivers and other low-level software for a modern microcontroller is truly vast, which makes software libraries all the more important.

More and more tech companies have been taking advantage of the seemingly underdeveloped field of services built around maintaining IoT systems. These companies belong to the third type of big players in today's IoT market. The service they usually provide is mainly centered around a means of data storage and manipulation for customers in the form of what is commonly called a cloud service.

Amazon and its AWS (Amazon Web Services) is a good representative of a company offering such a business model. In order to have a good customer uptake, it is very important to provide a high quality of service. This is related more to the application side of the service, such as efficient storage, good realiability and availability, data visualization, database maintenance and so on, and less so the underlying hardware and lower-level software sides. Amazon is also active in lower levels of the software hierarchy, though - since 2017, Amazon has taken stewardship of the FreeRTOS project and now offers a real-time operating system containing libraries and functions which can be integrated into AWS. [1] Other providers of cloud services include Microsoft Azure and Google Cloud.

1.1.3 Economical Considerations and Initial Cost Tradeoffs

A typical IoT or home automation company will put more emphasis on certain hardware and software requirements as its size changes. Smaller companies with less developers and a smaller customer base will generally select hardware from a big silicon manufacturer. This is because such manufacturers typically offer a better developer support.

The unit price for a bigger producer's better-supported chip might be higher, but it is less important for a smaller company's budget, as it might not sell a significant enough number of products for the slightly higher initial costs to matter.

On the other hand, for companies who move a relatively high amount of end products and operate on a bigger scale, the initial unit price might be more important than the manufacturer's product support capability. It might often be cheaper to simply hire more developers to battle issues and problems that present themselves during the development of a big product than to suffer from high initial hardware costs.

1.1.4 Embedded vs General-Purpose

This approach can also be applied and seen when it comes to developing a design of a particular IoT system. Nowadays, the rate of integration is rising and single-board general purpose computers, such as the Raspberry Pi, its various clones and devices based on a similar concept, are widely commercially available.

There is a lot of competition in this field, which means that prices are relatively low, which is a good trend for IoT companies. Single-board computers of today usually offer a complete solution of a CPU, power module, I/O functionality and peripheral tools that would in the past need to be sourced and assembled from many singular parts besides just the microcontroller.

For a smaller IoT company or start-up, the single-board computer approach offers a tremendous upside; the usual use of a conventional Linux distribution in such a device means that lots of different drivers are continuously being supported and bugs are often quickly patched. This means that using a single-board computer is actually more cost-effective in the long run, as less work needs to be expended to write low-level software.

1.1.5 Future Outlooks

The spread of IoT services has been exponential and has even seen some unexpected acceleration in recent years. In 2018, the number of devices connected to AWS rose by 49% over the preceding 12 months, and Azure grew by an even more staggering 93%. The number of connected devices that are in use exceeded 17 billion in mid-2018, with 7 billion of those being IoT devices, and is projected to exceed 34 billion by 2025 (21 billion of those being IoT). [2]

The development and nearing worldwide deployment of fifth generation networking (5G) communication technology will probably accelerate the growth of the IoT market even more.

Its promises of high bandwidth and, more importantly, extremely low latencies, will be very important in this field. It is projected that 3.5 billion IoT devices will be connected via cellular networks alone in 2023. [3]

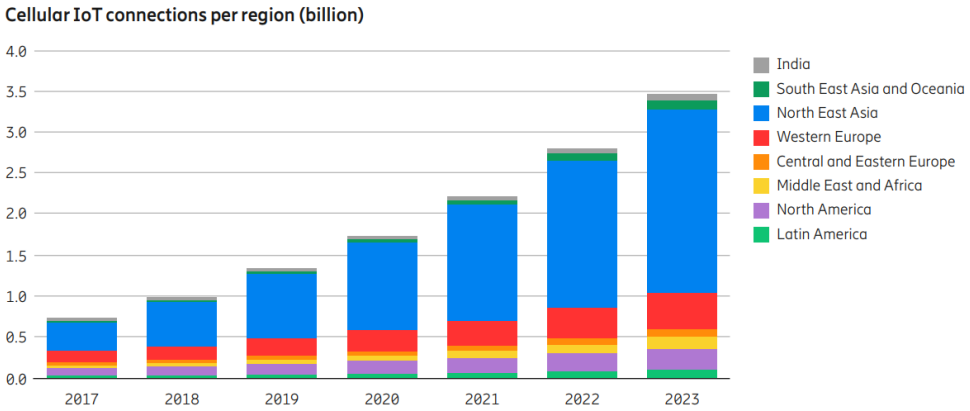

Figure 1.1: Cellular IoT connections forecast for 2023 [3]

As mentioned before, the period of moral obsolescence of today's both IoT and non-IoT devices is lengthening. Another fact is that semiconductor manufacturing technology advancements are slowing down - in 2019, 7 nanometer technology is slowly becoming the standard, with 5nm proposed for 2020 and 3nm for 2024. [4]

It is entirely possible that the market will see a shift with big companies switching to the field of providing services mentioned above - it seems that there is a relatively untapped profit potential, judging from the insane market growth in the recent past, present and near future.

Chapter 2

Hardware

In this chapter, first an overview of the iQtec home automation system is provided. Then the details of iQtec's existing hardware are introduced, along with its strong and weak points. In another section, the proposed new hardware gets presented and its specifics are discussed.

2.1 iQtec Home Automation System

iQtec is a system used for technological process control and data collection. It is highly modular and can be used in a wide variety of applications, ranging from controlling the environment of a single apartment or house, all the way to gathering data and regulating of large industrial objects. Its main usecases are as follows:

- Temperature regulation
- Heating, ventilation and louver control
- Lighting control, including dimmers and RGB scenes
- Electrical outlet control
- Energy consumption monitoring (electricity, gas, water, etc.)
- Home security camera and intercom functionality

Each additional sensor or control module connected within this system has its own proprietary software driver, which enables it to get registered, initialized and - most importantly - take part in communication with the main system unit.

2.2 Existing Solution

iQtec's PLC, or programmable logic controller, is comprised of a logic board with an STM32F437ZIT6 microcontroller unit. This MCU is made by STMicroelectronics and is based on an ARM Cortex-M4 core operating at a frequency of 168 MHz. This MCU itself contains 256 kilobytes of RAM, two megabytes of flash memory, and a real-time clock (RTC) module. [5]

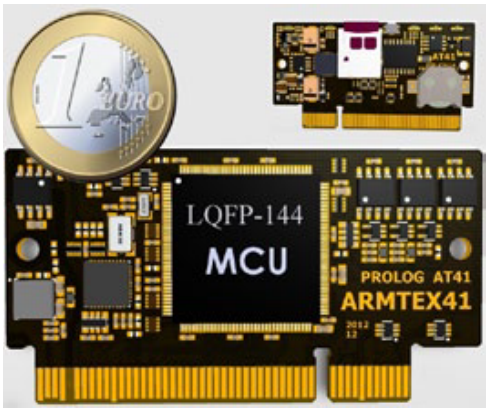

Figure 2.1: iQtec PLC [5]

There are also numerous peripherals inside the PLC [5]:

- A Ferroelectric RAM (or FRAM) module with a 128 kilobyte capacity connected on one of the MCU's SPI buses. This FRAM is used to store data that changes frequently, such as outputs from various measurement modules. The main strength of this FRAM module is its extreme read/write endurance - it is possible to overwrite every bit up to 10^{13} (or ten trillion) times. [6]
- An electrically erasable programmable read-only memory (EEPROM) chip, mainly used to store longer-term data, such as device initialization logs and error messages.

- A microSD card interface, which is primarily used to store service information or update the unit's firmware if needed. An industrial-grade microSD card is used in order for it to sustain temperature fluctuations reliably.
- Transceivers for many different communication protocols, such as RS-422, RS-485, CAN 2.0B, or Ethernet. These are used to communicate with the system's sensors and data collecting devices.

Most of the devices listed above are connected to the MCU via either of I^2C, SPI or UART-based serial interfaces.

2.2.1 Weak points

The weakest point of this PLC is the microcontroller unit. A massive drawback is its memory size. 256 kB of RAM is not a very comfortable amount to work with as a developer, especially when the code is powering a project which can potentially be quite extensive. There is a significant amount of effort that has been expended into designing the iQtec's software in an efficient way. Lots of time was put into memory optimization in the existing solution to make it run smoothly.

Another drawback is the relatively low computing power of the system's processor. A 168 MHz Cortex-M4 core still offers plenty of power for simpler usecases, but it has been showing signs of running out of breath in this application.

Steps had to be taken when developing the original iQtec codebase to reduce the memory footprint of some of its functions. These steps resulted in some unfavorable behavior of the system, which became apparent as the codebase grew and the system got more complex. Some of these drawbacks will be explained in detail in chapter three.

2.3 New Solution

2.3.1 Hardware Requirements

Research had to be conducted in order to choose a processing unit to act as a basis of the project's hardware. To make sure the correct chip was chosen in the end, a list of system requirements was agreed upon and made up. These requirements took into account the weak points mentioned in the previous section, as well as some parameters of the old solution that should be carried over and executed in a similar or better fashion.

Multi-Core Processing

The decision to create this project to use a traditional Linux operating system meant that the new CPU should be able to comfortably switch between lots of threads running concurrently. Therefore, a chip with multiple cores should be chosen.

Computing Power

This requirement also stems from the usage of a more complex operating system. To put this requirement into a context of numbers, the DMIPS metric should be explained first. DMIPS stands for Dhrystone Millions of Instructions Per Second, where Dhrystone is the name of a synthetic computing benchmark that quantifies a processor's performance in an accurate manner. A metric of "specific DMIPS" can be defined as the amount of DMIPS achieved for every 1 MHz for a given chip's clock rate.

The old project's processor has a DMIPS value of 210, or a specific DMIPS of 1.25 with a clock rate of 168 MHz [5]. A requirement was defined to provide at least five times the DMIPS of computing power with the new chip. This is quite generous, as it should be possible to beat 1000 DMIPS with almost any chip which already satisfies the first requirement and runs at a reasonable clock speed. For example, if a hypothetical dual-core Cortex-M4 chip existed, it would only have to run at a clock speed of 400 MHz to satisfy this requirement.

Memory

As mentioned in the previous section, a low amount of RAM is quite limiting when developing software for embedded systems, and can be downright painful as the project snowballs with more features and an expanding codebase.

To make sure there is enough system memory to satisfy the project's needs even as it gets improved in the future, while at the same time leaving some extra for the operating system itself, it was decided that at least 512 megabytes of memory should be available in the new hardware solution. This amount trumps the original 256 kilobytes by a huge margin, while being more than reasonable in today's state of the art.

Interfaces and Storage

The original system's PLC offers a relatively large array of communication interfaces. Five UARTs, three SPI bus interfaces, two I^2Cs, two USB 2.0 ports, an Ethernet port and more [5]. The new system should provide a comparable amount of these interfaces, with Ethernet and UART being the most important ones. There should also be options available to expand the IO ports with more functionality.

A microSD card slot is also an important requirement, as it stores crucial service information, as mentioned in section 2.1, though this is not necessarily a feature of the processing unit itself. A dedicated storage device (other than the microSD card) would also be welcome - the original design only incorporates a two megabyte flash storage, which is far being optimally sized.

2.4 Candidates

2.4.1 Variscite DART-6UL System-In-Module

The Variscite DART-6UL is one the platforms selected for closer examination. It includes a processor based on NXP's i.MX 6UltraLite family and contains an ARM Cortex-A7 core clocked at up to 900 MHz and 1 gigabyte of RAM. Its features also include two Ethernet connections, dual CAN, I^2C, SPI, UART and SD/MMC interfaces. The DART-6UL is optimized for low power consumption and is also relatively small in size, with the whole module's dimensions being only 25 by 50 millimeters. The DART-6UL also supports Linux as its operating system of choice. [7]

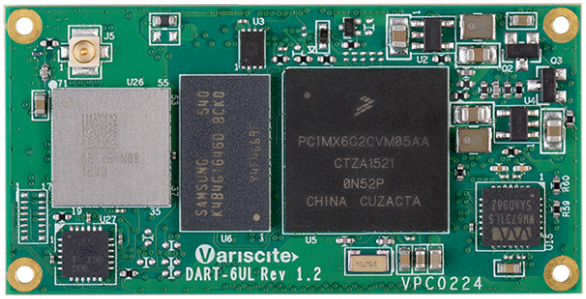

Figure 2.2: Variscite DART-6UL [7]

The Variscite DART-6UL satisfied demands for available storage devices and communication interfaces. It only uses a single Cortex-A7 core though, which might not be enough to handle the needs of the project when combined with possible future extensions. The lack of multiple cores also hurts its computing power - while its DMIPS value of about 1760 is far higher than the original system's 210, there are multi-core variations of the Cortex-A7 which would help even more.

A big advantage of this system-in-module is the fact that Variscite also produces and sells a 4G LTE modem module, which will be incorporated into the project at some point in the future. Combining these two devices could prove to be favorable, especially when it comes to development support.

2.4.2 STMicroelectronics STM32MP1-Series microprocessor

In February of 2019, STMicroelectronics announced a new series of microprocessors, the STM32MP1. This family of microprocessors uses a single or dual ARM Cortex-A7 core in combination with a Cortex-M4 core, clocked at 650 and 209 MHz, respectively. This enables the STM32MP1 to support multiple and flexible applications and achieve best performance and power figures. Up to 1 gigabyte of DDR3 RAM is available. Communication interface equipment is very good, with six I^2C, up to 8 UART, six SPI, three SD/MMC and a single Ethernet connection available. As for operating systems, both Linux and Android are supported. [8]

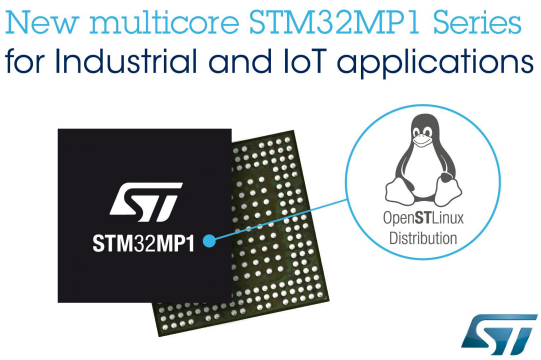

Figure 2.3: STMicroelectronics STM32MP1 [9]

The STM32MP1 satisfied all of the requirements defined in the previous section. It offers two Cortex-A7 cores clocked at 650 MHz, which comply to the computing power requirement with a DMIPS value of 2470. A big plus is the amount of communication interfaces available, as well as the number of storage devices available. A discovery or evaluation kit was not available at the time of this research, so the usage of the STM32MP1 was not pursued further. It is also very new and fresh on the market, which could have negative implications as it is currently relatively unproven, though they are not factual as of writing this thesis.

2.4.3 Octavo Systems OSD3358 System-In-Package

The last candidate that made the short list is Octavo Systems' OSD3358 System-In-Package (SiP). The OSD3358 itself integrates multiple devices in order to make it a system that is completely ready to run.

It integrates a Texas Instruments Sitara AM3358 processor, a TI power management system, up to 1 gigabyte of DDR3 memory and lots of passive components, all while keeping everything inside a single reasonably-sized 27x27 mm package. With this level of integration, the OSD335x Family of SIPs allows designers to focus on the key aspects of their system without spending time on the complicated high-speed design of the processor/DDR3 interface or the PMIC power distribution. It also reduces the overall size and complexity of the design and the supply chain. [10]

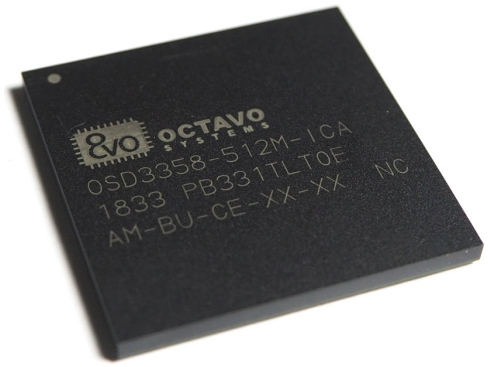

Figure 2.4: Octavo Systems OSD3358

The OSD3358 offers an Ethernet connection, two USB 2.0 ports, access to multiple CAN, SPI, UART and I^2C peripherals, a SD/MMC interface and an optional integrated eMMC storage device. [10] It is capable of running Linux, Android or an RTOS (real-time operating system) as far as operating systems go.

The Octavo OSD3358 also satisfies most of our requirements. It boasts solid computing power - its Cortex-A8 clocked at up to 1 GHz offers a DMIPS value of 2000 with a specific DMIPS of 2 per MHz - a value slightly higher than any of the other candidates. Peripheral interface connections are plentiful and a combination of eMMC and microSD card support is a big plus, as well as its size.

In the end, the OSD3358 was picked as the basis of this thesis's project. The main factors in this decision were its dimensions, its Sitara AM3358 processor, integrated eMMC and good peripheral connection capabilities. Its theoretical performance was also important in the decision, though it does only use a single-core processor.

Note on development board used

An Octavo OSD335X-SM-RED evaluation board was ordered and acquired in preparations to start the prototyping phase. Issues appeared with Ethernet drivers in the Linux distribution of choice in the early stages of prototyping. The Octavo-supplied evaluation board uses an Atheros AR8035, whose drivers were not functioning properly at the time. This meant that in the end, most of the prototyping was done on a BeagleBone Black board.

In the final product, it is likely that the Octavo SiP will be used, and a different Ethernet adapter will be used instead of the AR8035 located on the Octavo OSD355X-SM-RED.

2.4.4 BeagleBone Black

Due to issues with the Octavo OSD355X-SM-RED evaluation board early on in the development of this project, the decision was made to switch to a BeagleBone Black single board computer. The BeagleBone Black is extremely similar to the Octavo evaluation board, but there are some differences.

The key difference is that this board does not in fact run the Octavo OSD3358 chip - the main CPU of the BeagleBone Black is TI's Sitara AM3358, i.e. the exact same CPU found inside the OSD3358. This has no practical consequences on this project, as these CPU solutions only differ in power delivery circuitry, which does not affect this project in any way. The BeagleBone uses an SMSC LAN8710A Ethernet adapter, which did not exhibit any driver problems during development.

Another notable difference between the Octavo evaluation board and the BeagleBone Black is the amount of RAM, of which the latter only has 512 megabytes.

The BeagleBone Black is actually a very good board to use for this project. It is widely commercially available and relatively inexpensive (the cost of one unit at the time of writing is about $55). This means that the BeagleBone Black is very favored among developers and hobbyists alike. There is a big community support for this board, both from users and the manufacturer.

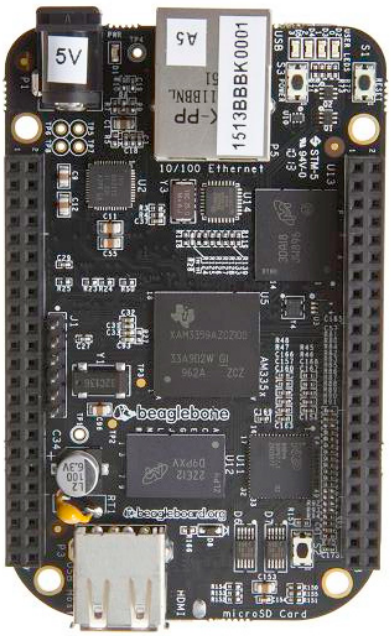

Figure 2.5: BeagleBone Black [11]

The main features of the BeagleBone Black are as follows [11]:

- TI Sitara AM335x 1GHz ARM Cortex-A8 processor
- 512MB DDR3 RAM
- A 3D graphics accelerator and a microHDMI port, which can be utilized to use the BeagleBone Black as a traditional single-board computer
- USB client and host connections
- A microSD card slot, as well as a 4 gigabyte eMMC memory, which allows it to be shipped with a preinstalled OS to provide out of the box usability
- Ethernet, two 46-pin input/output headers
- 4x UART, 2x SPI, 2x I2C, 2x CAN bus peripheral drivers

2.4.5 Texas Instruments Sitara AM3358

The AM335x family of microprocessors, of which the AM3358 is a member, are based on the ARM Cortex-A8 processor core and are enhanced with image, graphics processing, peripheral and industrial interface options. These devices also support high-level operating systems (HLOS). [12]

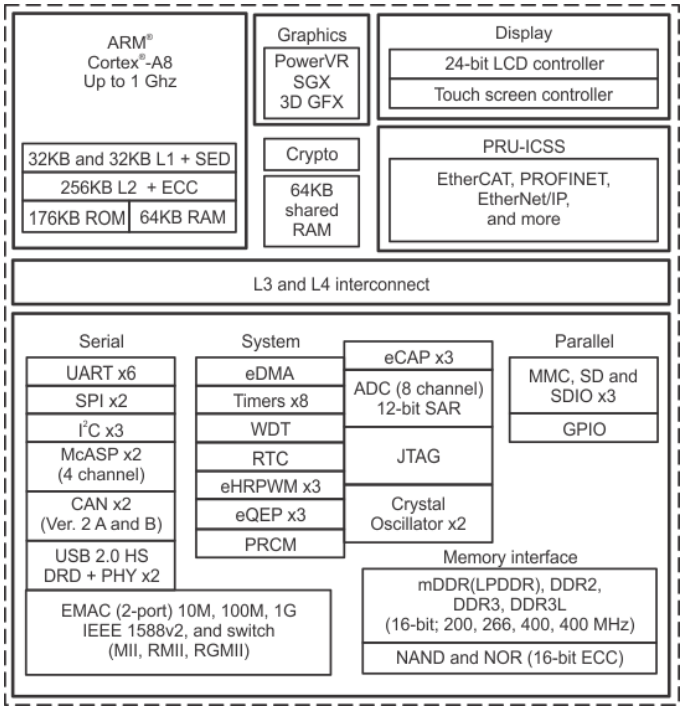

Figure 2.6: Functional block diagram of the TI AM3358 [12]

The AM335x family chips also contain two additional 32-bit RISC microcontrollers, called the Programmable Realtime Units, located on the SoC in the PRU Subsystem and Industrial Communication Subsystem, or PRU-ICSS. The PRU-ICSS is separate from the ARM core, allowing independent operation and clocking for greater efficiency and flexibility. The PRU-ICSS enables additional peripheral interfaces and real-time protocols. The programmable nature of the PRU-ICSS, along with its access to pins, events and all system-on-chip (SoC) resources, provides flexibility in implementing fast, real-time responses, specialized data handling operations, custom peripheral interfaces, and in offloading tasks from the other processor cores of SoC. [12]

The AM3358 CPU being used here runs at a clock speed of 1 GHz.

2.4.6 Lapis Semiconductor MR45V100A

The existing PLC uses a 128 kB FRAM chip to store data that changes very frequently and is overwritten constantly, such as temperature or energy consumption measurements, or other information of similar character.

This prototype uses an MR45V100A FRAM chip for this very same purpose. It is a one megabyte unit which should be good for trillions of overwrites per bit. It can be used in temperatures ranging from -40 to +85 degrees Celsius, which will be enough for our use. [6] It is connected to the CPU via a serial peripheral interface (SPI).

A picture of the FRAM chip as installed on the prototype can be seen in figure 2.7.

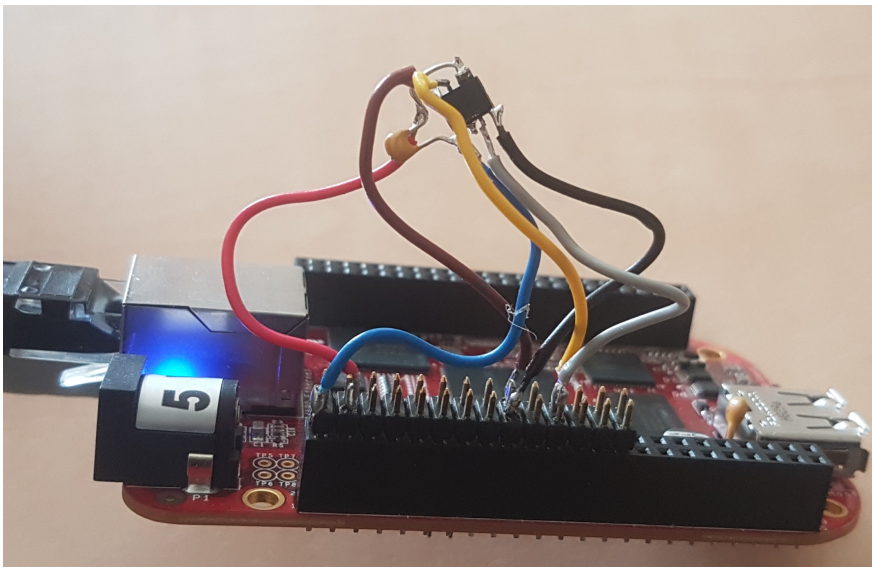

Figure 2.7: MR45V100A ferroelectric RAM

Chapter 3

Software

The software side of this thesis is arguably the part that needed improvement the most.

The iQtec functionality itself and all the system device (i.e. hardware other than the PLC itself) drivers will not need reimplementing or any drastic changes, as the system devices are not a part of this prototype, rather just peripherals connected to it.

The microcontroller-esque nature of the original PLC design is not ideal in today's state of the art, as outlined in 1.1 and 2.2. As the scope of the original project rose, some sub-optimal routes had to be taken in order to ensure proper functionality while staying inside the constraints of the hardware and system resources.

In this chapter, some of these solutions will be explained. New approaches will be discussed and introduced, and a general comparison will be provided between the old and new solutions.

3.1 Operating System

3.1.1 Existing Solution - FreeRTOS

The old PLC uses FreeRTOS as its operating system of choice.

FreeRTOS is a market leading real time operating system, or RTOS for short, from Amazon Web Services that supports more than 35 architectures. It is professionally developed, strictly quality controlled, robust, supported, and free to embed in commercial products without any requirement to expose your proprietary source code. FreeRTOS has become the de facto standard RTOS for microcontrollers by removing common objections to using free software, and in so doing, providing a truly compelling free software model. The FreeRTOS kernel was originally developed by Richard Barry around 2003, and was later developed and maintained by Richard's company, Real Time Engineers Ltd. FreeRTOS was a runaway success, and in 2017 Real Time Engineers Ltd. passed stewardship of the FreeRTOS project to Amazon Web Services (AWS). [1]

FreeRTOS has been used in iQtec's PLC because of its good viability when used in a microcontroller-based system. FreeRTOS is generally designed and created to have a small memory footprint and a small processor overhead. It also supports multi-threading. There are none of the more advanced features typically found in operating systems like Linux or Microsoft Windows, such as device drivers, advanced memory management, user accounts, and networking. The emphasis is on compactness and speed of execution. [13]

The second to last point mentioned in the previous paragraph is one of the bigger downsides of using FreeRTOS here. Networking and device drivers have to generally be "outsourced" to external libraries, or written from scratch. To avoid doing this in this thesis's prototype, it was decided to not use an operating system of this nature, and instead go for a traditional Linux distribution, which offers the "luxury" of having network and device drivers ready to go, as well as a good memory management system.

3.1.2 RTOS in General

A big strength of using a lightweight operating system such as FreeRTOS can be found in its name - it is a true real time operating system.

A processor core can only run a single thread of execution at a time. To provide multi-tasking, an operating system has what is called a scheduler, which is responsible for deciding which program to run when, and provides the illusion of simultaneous execution by rapidly switching between programs. [14]

The type of an operating system is defined by how the scheduler decides which program to run when. For example, the scheduler used in a multi-user operating system (such as Unix) will ensure each user gets a fair amount of the processing time. Some operating systems prioritize tasks that interact with the user, which helps with user experience and perceived responsiveness. [14]

The task scheduler in a real time operating system provides a deterministic execution pattern. This is very useful in embedded systems, as these systems have real time requirements. A real time requirement is one which specifies that the system must respond to a certain event within an expected and strictly defined time frame. Real time operating systems generally guarantee real time requirements because the system's task scheduler runs in a deterministic manner, and thus its behavior can be predicted. [14] A typical way of providing determinism and predictability is the use of thread priorities. The highest priority task wanting the CPU always gets the CPU within a fixed amount of time after the event waking the task has taken place. On an RTOS, the latency of a task only depends on the tasks running at equal or higher priorities. On a normal operating system, these latencies depend on everything running in the system at a given point in time, which makes it very difficult to meet a task's deadline reliably. [15]

3.1.3 Combining RTOS and Linux Strengths - PREEMPT_RT

A traditional Linux-type operating system is not an RTOS. Many programs and threads run in parallel and typically, no real time requirements are specified nor guaranteed. If kernel code is executing when some event takes place that requires a high priority thread to start executing, the high priority

thread can not preempt the running kernel code, until the kernel code explicitly yields control. In the worst case, the latency could potentially be hundreds of milliseconds or more. [15] There is, however, a way to bring Linux's kernel closer to that of a real time operating system - The PREEMPT_RT patch.

PREEMPT_RT is a real-time kernel patch which can transform the Linux kernel into one that is fully preemptive. In addition to faster response times (and maybe more importantly), it removes all unbounded latencies. An unbounded latency means that the amount of delay that can occur is dependent on the situation. [16]

In a normal Linux kernel, a total of four preemption models are implemented which alter the kernel's behavior. These models provide tradeoffs between system throughput and latency, with the "No Forced Preemption" model having the best throughput at the cost of worse latency, while the "Low Latency Desktop" model is at the other end of the perceived spectrum. The PREEMPT_RT patch adds two more preemption models, one of them dubbed "Fully Preemptible Kernel." This flavor of the patch makes all kernel code preemptible, save for a few selected critical sections. Several substitution mechanisms, such as special real-time mutexes, are implemented to reduce preemption-disabled sections of code to a minimum. This model is the one that truly provides the kernel with real-time behavior. [17]

3.1.4 Real-Time Thread Scheduling

In an ordinary Linux operating system (or any other general purpose OS), tasks need to share CPU time, as there are orders of magnitude more tasks than available CPUs. Waiting tasks are usually dispatched in order of decrasing priority. If two or more tasks' priorities are identical, these kinds of ties are broken according to task scheduler policies. [18] All scheduling is preemptive - if a task with of a higher priority becomes ready to run, the currently running task is preempted and returned to the list of waiting tasks. [19]

The standard task scheduler in Linux is called CFS, or "Completely Fair Scheduler." Internally, this scheduler is called `SCHED_OTHER`. It uses an attribute called the *nice value*, which in essence represents a task's priority level. The range of the nice value varies across different Unix systems, but on modern Linux, the range is -20 (high priority) to +19 (low priority). [19] This might seem counterintuitive when thinking about niceness as a priority, but what it actually represents is how "nice" a task is to other waiting tasks.

A very nice task will let other tasks skip ahead of it, while a task that is not nice will not.

A Linux kernel patched with PREEMPT_RT introduces more functionality when it comes to scheduling and priorities. There are two more task scheduling policies, as well as an additional priority system. One of the new policies is called `SCHED_RR`. When using this scheduler, tasks of equal priority alternate using a simple round-robin scheme. The other new policy is `SCHED_FIFO`. As its name might suggest, tasks with equal priorities are dispatched in the order in which they were enqueued into the task queue. [18]

When using one of these new scheduling policies, threads gain the ability to have a new set of priority values. A static scheduling priority, or `sched_priority`, is what the scheduler uses in order to decide which task to dispatch next. `sched_priority` is not used in this decision at all when not using any of these two new policies (it is always set to zero for all tasks). Processes scheduled under these policies have a `sched_priority` value in the range of 1 (low priority) to 99 (high priority). This means that real-time scheduled threads will always have higher priority than normal tasks. [19]

3.1.5 Effects of the PREEMPT_RT Patch

To quantify how PREEMPT_RT actually affects performance in real-time applications, two tests were conducted. The first test's purpose was to measure the difference in latencies between a Linux system where the PREEMPT_RT patch wasn't applied, and a Linux system that had previously been patched.

In order to prepare the testing environment, an image of Debian 9.8 suited for the BeagleBone Black was sourced and flashed onto a microSD card. This system was then booted up unmodified, and the first batch of tests was run. After that, the system was patched with PREEMPT_RT, and the same tests were run again. Applying the PREEMPT_RT patch on the kernel of the BeagleBone Debian distribution is quite simple. The Debian distribution ships with a script that will take care of most of the patching process.

```
root@beaglebone:~# cd /opt/scripts/tools/
root@beaglebone:/opt/scripts/tools/# uname -r #verify kernel version
4.14.71-ti-r80
root@beaglebone:/opt/scripts/tools/# ./update_kernel.sh \
> --ti-rt-channel --lts-4_14
```

After the script finishes executing, a reboot is needed for the patch to take effect.

Verification of the model of preemption can be done very quickly - the configuration file `/proc/config.gz` can be decompressed and displayed with gzip, and all lines containing the word "PREEMPT" can be filtered with `grep`. A flag will be displayed for each of the preemption models. If the CONFIG_PREEMPT_RT_FULL flag is present and enabled like in the snippet below, the operating system's kernel is fully preemptible.

```
root@beaglebone:~# gzip -cd /proc/config.gz | grep PREEMPT
CONFIG_PREEMPT=y
CONFIG_PREEMPT_RT_BASE=y
CONFIG_HAVE_PREEMPT_LAZY=y
CONFIG_PREEMPT_LAZY=y
# CONFIG_PREEMPT_NONE is not set
# CONFIG_PREEMPT_VOLUNTARY is not set
# CONFIG_PREEMPT__LL is not set
# CONFIG_PREEMPT_RTB is not set
CONFIG_PREEMPT_RT_FULL=y
CONFIG_PREEMPT_COUNT=y
...
```

3.1.6 Cyclictest

The Cyclictest utility was used to measure the systems' latencies. Cyclictest runs a non real-time master thread which starts a defined number of measuring threads with a defined real-time priority. The measuring threads are woken up periodically with a defined interval by an expiring timer (cyclic alarm). Subsequently, the difference between the programmed and the effective wake-up time is calculated and handed over to the master thread via shared memory. The master thread tracks the latency values and prints the minimum, maximum, and average latencies after each iteration. [20]

In order to put load on the system while Cyclictest was running, the `stress` utility was used. `stress` is a very simple workload generator which can put a configurable amount of load on the CPU, memory, I/O and disk system. [21] It was configured to run 64 load tasks on the CPU and I/O system, as well as 4 tasks on the memory system, each randomly allocating and deallocating 64 megabytes of memory.

Cyclictest was launched and configured to run for 1,000,000 iterations with a thread timer running at a period of 1 ms. Cyclictest's priority was set to 99 (the highest possible priority). The exact same command was used while testing both the unpatched and patched kernels - the priority setting had no effect on the unpatched system. The output of Cyclictest consists of three numbers - the minimum, average and maximum values of thread latency in microseconds. Ideally, these three values should be as close as possible to each other, with the maximum not being very high relative to the average.

A typical output of Cyclictest on a non-RT-patched system looks like the following snippet:

```
root@beaglebone:~# stress --cpu 64 --io 64 --vm 4 --vm-bytes 64M &
root@beaglebone:~# ./cyclictest -t1 -p 99 -n -i 1000 -l 100000

policy: fifo: loadavg: 123.82 77.32 58.20 133/287 1796
T: 0 ( 1796) P:99 I:1000 C: 1000000 Min: 23 Act: 45 Avg: 44 Max: 421
```

You can see that the maximum latency gets quite wild, at about 10x the value of the average. This would not be acceptable in a real-time system.

The table in 3.1 displays the results of this first test done on a clean installation of BeagleBone Debian, and a PREEMPT_RT-patched version of the same system. You can see that using a real-time kernel does indeed bring significant improvements in performance and reduces latencies drastically.

	Min. Latency [μs]	Avg. Latency [μs]	Max. Latency [μs]
Debian 9.8, unpatched	23	44	421
Debian 9.8, PREEMPT_RT_FULL	16	24	49

Table 3.1: Comparison of Cyclictest results between unpatched and RT-patched systems

3.1.7 A Custom Threaded Benchmark

The second test that was executed was in essence quite similar to the first test. A more real-life source code was written (some of which would later be used in the ported code during the implementation phase of the thesis).

The purpose of this code was to spawn an independent thread with a high priority, which would keep a timer alive and, when the timer fired, would raise a POSIX semaphore structure so that other threads knew a set amount of time had passed since the last firing. To put it very simply, it is an emulation of a hardware timer running at a configurable period. The actual implementation calls for a 25 millisecond timer, so that value is what was used to benchmark the latencies in this test.

When running on the patched kernel, the thread priority was again set to 99, and the schedule policy was set to `SCHED_FIFO`. When running on the unpatched kernel, these settings were not applied as they are not supported.

The only task of the so-called "timer thread" is to wake up after catching a signal, set the timer again, and raise a semaphore. This whole cycle was timed using high-precision clocks in C's `time.h` library.

During testing, the system was put under load in the same manner as in the first test; using the `stress` utility. Each test was comprised of 8,000 iterations of a 25 millisecond timer, which means that one test instance took about 200 seconds to run.

The results of this test can be seen in figures 3.1 and 3.2. What they represent is, simply put, the overhead (in microseconds) of the implemented timing mechanism. Both figures also include a line representing the mean overhead values.

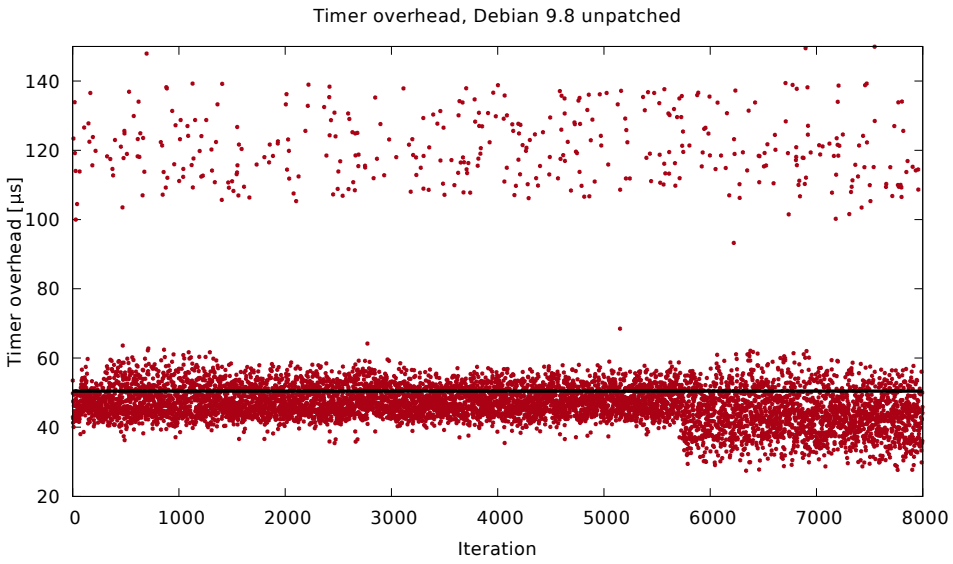

Figure 3.1: Timer overheads on non-patched kernel

The results are somewhat expected; the unpatched kernel performs worse, its mean overhead value is 50.44 μs and 90 % of measurements fall into an interval of $(37.375; 62.216)$. The outliers are rather extreme, and there were three occurrences of overheads larger than 1500 μs. Even though microsecond-precision is probably not needed in a 25 ms timer, a 1.5 ms delay is truly unacceptable.

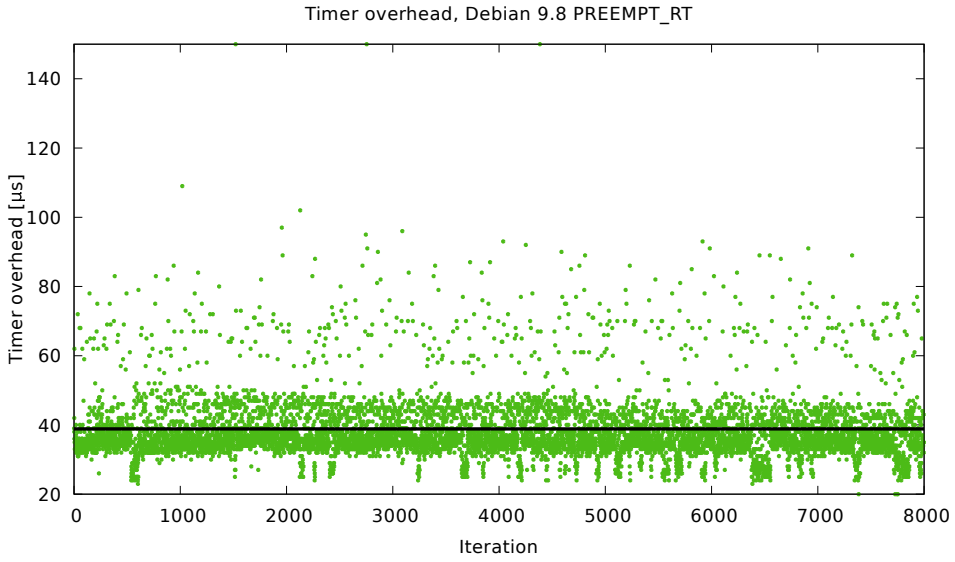

Figure 3.2: Timer overheads on PREEMPT_RT-patched real-time kernel

The PREEMPT_RT-patched kernel performed significantly better. Its mean timer overhead was only 38.86 μs, and 90 % of values measured fell into an interval of $(28.791; 53.085)$. Outliers were also less frequent and their values were not as high as with the unpatched kernel - just five iterations saw an overhead of more than 100 microseconds (0.0625%), versus a much higher 392 occurrences (4.9%) of the same during benchmarking of the unpatched kernel.

3.1.8 Choosing a Particular Operating System

After running these tests and consulting with the project supervisor, an agreement was reached to use the PREEMPT_RT-patched version of the official BeagleBone Debian distribution to conduct the rest of this thesis's experiments.

We agreed that while it still might have some performance reserves that could be improved by using a more "lean" operating system, this distribution has a lot to offer to a software developer. Things such as the system's automatically-configured virtual Ethernet connection over its USB port (which allows the user to connect to and transfer data to and from the device without a network), a reasonably well-written documentation and active support forums, as well as the fact that this very operating system is recommended and supported by Texas Instruments, the company making the CPU used in this thesis, were factors that helped make this decision even easier.

3.2 Filesystem

As discussed before, iQtec's original PLC uses FreeRTOS as its operating system of choice. Being a bespoke RTOS, its main useful parts are the task scheduling kernel, a means for inter-process communication (IPC), thread synchronization and timers. [22] When used as is, it provides little more functionality. FreeRTOS has to be statically linked with any user-supplied compiled source code to produce a single executable, which is then booted by the system directly (or using a bootloader). Most real-time operating systems including FreeRTOS can not load and unload code dynamically from a file system, just like a regular general-purpose operating system can. Code is usually loaded at start-up and runs until power-down. [23]

3.2.1 Existing Solution - FatFs

Programs running on microcontrollers don't usually have any filesystem capabilities. What these entail is the creation, moving and deletion of files and directories, reading and writing binary data to and from files, and communication with external storage devices, such as SD cards. The existing PLC incorporates all of this functionality, and uses FatFs to do so.

FatFs is a generic FAT filesystem module for small embedded systems, which is written in C and is very platform-independent. It can be incorporated into small microcontrollers with limited resources, as it boasts a very small memory footprint. [24] Some of FatFs's other features include [25]:

- Windows compatible FAT file system
- Support for multiple volumes (both physical drives and partitions), up to 10
- Long file name support in ANSI/OEM or Unicode
- Volume sizes of up to 2 terabytes (FAT32 specification when using 512 byte sectors)
- Support for RAM disks

3.2.2 File Access Slowdowns

One of FatFs's main uses in iQtec's system is communication with the microSD card. The microSD card's purpose is to store service information, new firmware versions when an update is available, error logs and a partial copy of the system's current program state. There can be up to hundreds of files on the card at a given point in time. The microSD card is usually accessed every 25 milliseconds, so it does see some load almost constantly. One of the drawbacks of using the FAT filesystem is the fact that file operations can get very slow in directories containing more than a few hundred files. This concerns not only write and read operations on files, but simple file accesses are affected, too.

This was reportedly a problem at some point in iQtec's system history - file access times would apparently take about 5 milliseconds from start to finish when only about twenty files were present in a given directory, but this access time would rise linearly up to unusable levels. What this means is that the time complexity of searching within a given directory is bounded by $O(n)$, where n is the number of files in the directory. With close to a thousand of files, access times were reaching upwards of 40 milliseconds. This is clearly unacceptable in a case where the filesystem has to be accessed every 25 milliseconds. A workaround was deployed where a larger directory's files were split into subdirectories by the first characters of their filenames. This brought a little bit of extra complexity into the code, but helped mitigate the file access time issue somewhat.

3.2.3 Quantifying Slowdowns

A simple program was created to test this behavior. The application would start with an empty directory and incrementally create files of random sizes. After each file was created, the application would select an existing filename at random and open the corresponding file. The access time was measured and outputted for every iteration. After the completion of this test, the directory would get scrubbed of all created files and the application would quit. The testing would terminate after 1000 files were created, which was more than enough to illustrate the point.

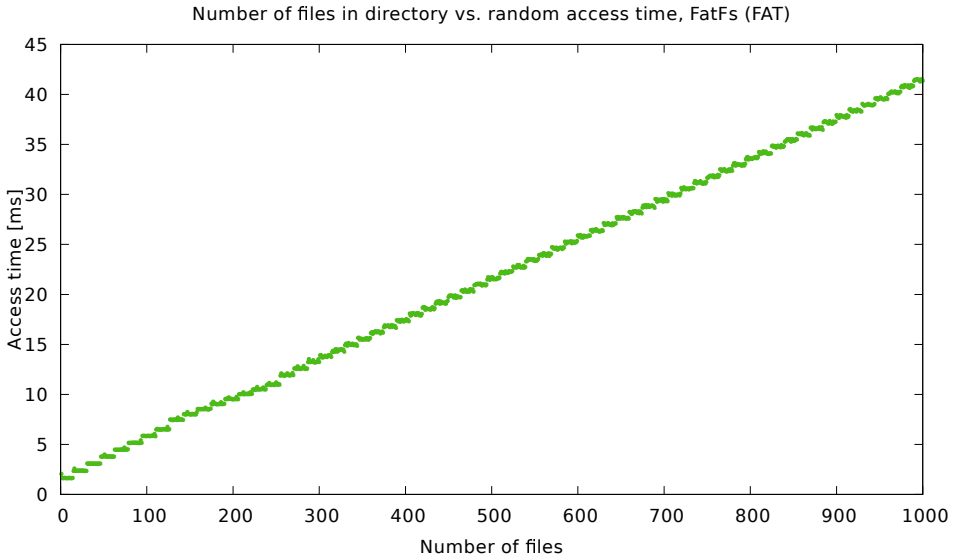

Figure 3.3: Random file access times depending on directory population, FatFs

This test was conducted numerous times and output logs were then analyzed. Typical results can be seen in figure 3.3. The data looks incredibly well-behaved - as iterations go on, the access times stay almost constant and only rise once every 16 iterations. The output looks almost like a plot of a step function. This is almost certainly caused by a construct inside FatFs that possibly reallocates the directory table in increments of 16 records to save space. A trend line is not included in this graph as it is fairly apparent that file access times rise in a linear manner as the number of files in the tested directory rises. The equation for the trend line is, according to Microsoft Excel, $f(x) = 0.04 \cdot x + 1.625$, which indeed corresponds to the theoretical access time complexity of $O(n)$.

The test confirmed the existence of one of FAT's weaknesses - directories are merely unsorted arrays of filenames, so a directory searching function simply goes through the entries one at a time, until it finally finds (or doesn't) the correct entry. This is not a problem for directories that are relatively sparse, but gets more apparent as a directory gets populated with many files. [26] The FAT system is not solely to blame, though. FatFs is a very lightweight and simple implementation of a FAT filesystem, which certainly does not help with speed and efficiency. The fact that a 168 MHz-clocked microcontroller is used as the hardware is another factor that surely contributes to the filesystem's slowness.

3.2.4 Sudden Shutdown Behavior

Another weakness of the existing solution is its relatively low tolerance against sudden shutdowns. The FAT filesystem is not inherently power fail safe. This means that if a power loss happens during file or directory updates, incorrect file metadata updates may happen. This produces garbage data, which in turn can cause a filesystem corruption. [27]

In FatFs, the filesystem is only updated once a file that is being updated is closed. Data loss due to sudden shutdown or media removal can be somewhat mitigated by periodically flushing a file that is opened for an extended period of time. The `f_sync()` function provides means to do this; only the data since the last synchronisation is lost after a sudden shutdown. After a search through the iQtec codebase, it seems as though this function was not being used at all, which definitely contributed in the system's intolerance to shutdowns.

3.2.5 New Solution - ext4

Since the decision was made to use a traditional general-purpose operating system in the form of a Linux distribution (Debian 9.8 for the BeagleBone Black), there will be no need to use an external filesystem solution. The operating system is installed on a volume using the ext4 filesystem, which is supported by the Linux kernel natively. This brings about numerous advantages over the formerly used FatFs.

One such advantage is the fact that ext4 is a journaling filesystem. This has tremendous implications regarding its sudden shutdown tolerance.

Journaling file systems are fault-resilient file systems that use a journal to log changes before they're committed to the file system to avoid metadata corruption. The journal is a special file that logs the changes destined for the file system. At periodic intervals, the journal is committed to the file system. If a crash occurs, the journal can be used as a checkpoint to recover unsaved information and avoid corrupting file system metadata. Figure 3.4 illustrates how a journaling system works. Ext4 also includes the ability to checksum the contents of the journal to make the journal more reliable. Another interesting fact about ext4 is its date resolution for file attributes - down to 1 nanosecond - although this will not be needed in this project. [28]

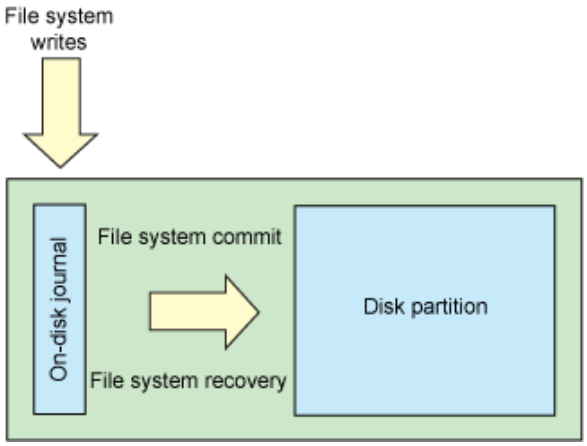

Figure 3.4: Schema of a typical journaling filesystem [28]

Another advantage that ext4 possesses over FatFs (and FAT in general) is speed. It does not suffer from the issue where file access times in a given directory would go up with the number of files in the directory. This is due to ext4's usage of efficient data structures to store directory entries. Where FAT used simple linear arrays to store such information, ext4 puts those into a special balanced tree. An advantage of balanced trees is that, while their implementation is more complex, they offer a search time complexity of $O(log(n))$, where n is the number of elements in the tree. [29]

A testing application was written to test the access times of files in an ever-expanding directory, very similar to the test executed on the FatFs-equipped old system. The inner workings of this test were identical to the test described in 3.2.3. This test was run on the new hardware and software configuration, i.e. a BeagleBone Black system running an installation of Debian 9.8.

The results of this test can be seen in a plot in figure 3.5. Other than the obvious speed-up (or rather lack of slowdown), please take note of the units used. The test showed that the ext4 solution used here was about two to three orders of magnitude faster than the formerly used FAT solution; where FatFs took dozens of milliseconds to access a file, ext4 only needs about 100 microseconds at worst.

The figure also includes a trend line this time, which, as it turns out, correctly reflects the time complexity by rising logarithmically with increasing file counts. According to Excel, the equation for the trend line's curve is $f(x) = 0.458 \cdot ln(x) + 73.27$, where $ln(x)$ is the natural logarithm of x.

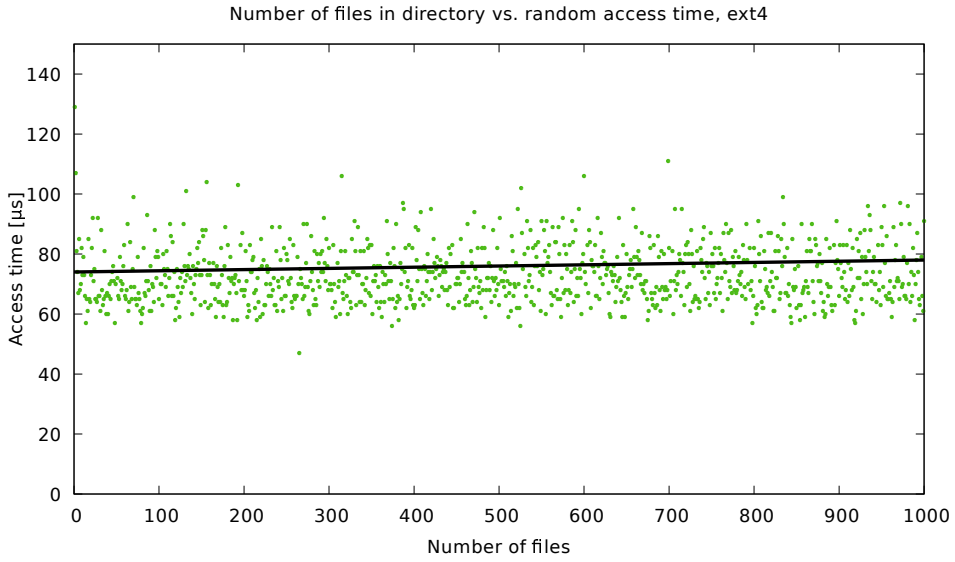

Figure 3.5: Random file access times depending on directory population, ext4

There is one more explanation for this massive performance gain, other than the usage of a superior, non-lightweight-implemented filesystem. It's also the fact that this test was executed on hardware that is much more powerful - the BeagleBone Black is equipped with a processor based on a very different microarchitecture (Cortex-A8 vs Cortex-M4), as well as having a significantly higher clock-rate (1 GHz vs 168 MHz).

3.3 IDE and Other Software Tools

3.3.1 Code Composer Studio

When developing source code for this master's thesis project, the integrated development environment (IDE) of choice was Texas Instruments' Code Composer Studio version 8.3. Code Composer Studio is based on the Eclipse open source software framework. This generally means that a user familiar with Eclipse will find the user experience of CCS to be very similar. The choice to use CCS was made because of its very good integration of various functionalities used in developing applications for Linux-based systems.

Figure 3.6: Code Composer Studio splash screen

3.3.2 Microsoft Visual Studio 2017

Microsoft's Visual Studio was also used for a portion of the development. iQtec's codebase is contained in multiple Visual Studio projects, and plans were made to integrate the Linux-deployed code into the existing projects. The correct platform would then simply be selected in Visual Studio.

Generally, individual functional components of this project were developed and debugged using Code Composer before being integrated into the iQtec codebase in Visual Studio.

3.3.3 Cross Compilation

During this project, development was done on ordinary desktop computers running the Microsoft Windows operating system. The target device, on the other hand, was an ARM processor running Linux. To bridge the gap between the development and target (or host) environments, a cross compiler is used. A cross compiler is a tool that allows the developer to compile source code on one platform and run it on a different platform.

The cross compiler used in this project was a part of the Linaro Toolchain. What's included in this toolchain, besides the GCC compiler and compiler libraries, is a set of tools that allow the inspection, debugging, and profiling of executables [30]. The Linaro Toolchain is maintained by the Linaro engineering organization and new versions are released periodically.

In order to compile on an x86-64 operating system and deploy onto an ARM system, the `5.5.0-2017.10-i686-mingw32_arm-linux-gnueabihf` package was used. This designation can be broken down into individual parts - `i686-mingw32` signifies the host environment, `arm` signifies the target architecture, `linux` is the target system, and finally `gnueabihf` indicates which ABI (application binary interface) is used during compilation (GNU in this case). [31]

3.3.4 Integration of the Cross Compiler

The integration of cross compiler toolchains into Code Composer Studio projects is incredibly easy. Once a toolchain is extracted and installed on the host environment, it is only a matter of adding its path into Code Composer's compiler settings, and finally selecting the desired compiler from a dropdown menu in a project's properties.

When it comes to Visual Studio however, the integration of the Linaro toolchain was a very different experience. As it turns out, it is impossible for Visual Studio 2017 to support third-party cross compilers. Linux targets are supported with a slight workaround, though. It is possible to set up Visual Studio for remote compilation for Linux C/C++ development. Since Windows 10 ships with the option of installing a Linux subsystem (called the Windows Subsystem for Linux, or WSL for short), it is possible to point Visual Studio to target WSL. This means that Visual Studio will compile source code "remotely," with the target system running on the same machine.

To take advantage of this workaround, it is then possible to simply obtain a version of Linaro toolchain that runs on x86-64 Linux and targets our ARM system, and finally point WSL to use that compiler instead.

What this means in summary is that Visual Studio will "remotely" copy all source code onto the Windows Subsystem for Linux, compile it using a Linaro cross compiler targeting a system that is completely different from what it thinks, and then return the ready-to-deploy binary file back to the user in Windows.

The particular version of Linaro used for this workaround is `5.5.0-2017.10-x86_64_arm-linux-gnueabihf`. To verify that the entire cross compilation process works without issues, we can simply run the `file` command on the resulting binary and check that the target architectures are identical.

```
root@beaglebone:~# gcc hello.c -o hello_gcc
root@beaglebone:~# ./hello_gcc
Hello, World!
root@beaglebone:~# file hello_gcc hello_ccs hello_vs2017
hello_gcc:    ELF 32-bit LSB shared object, ARM,
              EABI5 version 1 (SYSV), dynamically linked, ...
hello_ccs:    ELF 32-bit LSB executable, ARM,
              EABI5 version 1 (SYSV), dynamically linked, ...
hello_vs2017: ELF 32-bit LSB executable, ARM,
              EABI5 version 1 (SYSV), dynamically linked, ...
```

Note: There is no difference between the terms "executable" and "shared object" in this context.

Chapter 4

Implementation Process

Chapters 2 and 3 provided an exhaustive description of both the old and new solutions when it comes to the hardware and software used. A consequence of the dramatic hardware and software changes is that a significant part of the codebase needs to be rewritten from scratch.

A portion of the codebase could simply be migrated from the old system to the prototype developed in this thesis. This particular portion consists of peripheral device drivers which are connected to the iQtec PLC via standardized communication protocols, such as SPI or other UART-based protocols. The functionality of these peripherals should remain unchanged provided the handling of communication on the aforementioned buses is implemented correctly.

A lot of the codebase became incompatible with the new system specifications. Some code depended on constructs that are not available or supported in the same way in this thesis's project, such as hardware timers and external real-time clocks. Other parts of the code were simply outdated or depended on technology that has been surpassed, like the filesystem - this issue was described in 3.2.

This chapter will provide to the reader an overview of what tasks needed to be accomplished in order to begin developing new functionality for this project. The process of enabling various peripheral interfaces connected to the CPU of choice, the Texas Instruments Sitara AM3358, will be described. Each system component that needed porting over to the new solution will be introduced in detail and the specifics of its reimplementation will be discussed.

4.1 The Linux Device Tree

The Linux system, when running on an embedded device, does not have a BIOS. In order to boot and configure the hardware device it is running on, it uses files located on the filesystem that describe the machine's hardware. Every type of embedded Linux device has its own unique set of files to describe its platform hardware. Previously, modifications to the Linux kernel were applied each time Linux was booted from a new unique embedded device. As the popularity of ARM-based microprocessors soared, thousands of such modifications were implemented. Nowadays, ARM-powered embedded devices use *device trees* to describe hardware configurations instead. [32]

The data contained within a device tree describes the type of CPU, memory, input and output pins and other devices connected to it (such as Ethernet, external storage, SPI/I^2C buses, etc.). [33] Figure 4.1 illustrates a simple device tree structure. [34]

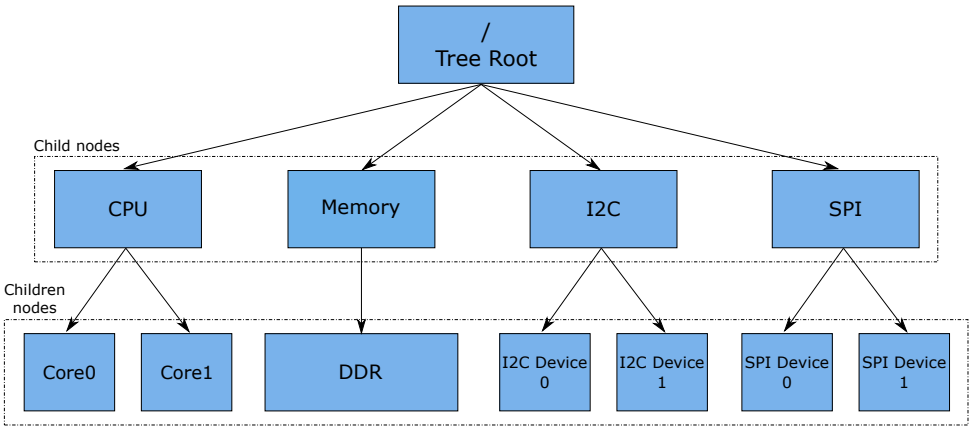

Figure 4.1: Simplified diagram of a device tree structure (adapted from [34])

These files are used as part of the boot process. To modify the hardware that is enabled in the system, it is possible to decompile the full device tree, make changes, then recompile and deploy it, which is a rather tedious process. An alternative way of doing it is by using a device tree overlay. A device tree overlay supplies additional information about the hardware. It is simply merged with the full device tree by the bootloader (typically U-Boot) and then passed to the Linux kernel at boot-up. [32] A device tree overlay's binary form is called a *device tree blob*, or .dtb file. The human-readable format for device trees typically uses the .dts extension. A device tree compiler (DTC) is used to to transform between the two formats.

Note on documentation confusion

Previously, it used to be possible to load device tree overlays at runtime. A kernel mechanism called Capemgr (short for cape manager) was used to dynamically load overlays without the need to reboot the system, as long as the device tree blob file was prepared in advance. These overlays were originally used to support custom expansion boards (capes) for various development boards which used device tree-supporting kernels. [35] Though very convenient, this approach is no longer supported due to stability issues and is being phased out to be replaced with U-Boot overlays, which are more stable and well supported, though they do require a reboot. [36] This overhaul started in late 2017 (with kernel version 4.14), more than five years after Capemgr first started being used. This means that lots of documentation sources contain now-obsolete information and troubleshooting device tree-related problems requires a considerable amount of effort.

4.2 Enabling Interfaces

The Sitara AM3358 chip contains an abundance of peripheral interfaces. Approximately half of its terminals can multiplex up to eight modes of signal functions. There are many combinations of pin-multiplexing that are possible, but only a certain number of combinations, or IO sets, are permitted due to timing limitations. A table of all possible pin attributes can be found in the AM335x datasheet. [37]

The BeagleBone Black has two 46-pin headers of GPIO pins, 92 in total. Almost every one of these pins (except for ground and power supply pins) can be configured for one of up to 8 possible operating modes. This configuration can be changed by unloading and loading different device trees. The default configuration of the Debian distribution used in this project is unfavorable for our use - a significant number of pins are allocated for multichannel audio, LCD and HDMI output drivers, which means they cannot be used as traditional GPIO (general-purpose input/output) pins. These allocations can be freed by simply not including the HDMI (or other) functionality in the device tree.

As the operating system used here is supported by both Texas Instruments and BeagleBoard.org, it ships with various device tree overlays included, which enable the developer to pick and choose pin multiplexing sets, which in turn defines what interfaces will be available and enabled.

There is no need to recompile device trees, as there are numerous pre-prepared device tree blobs located in the `/lib/firmware/` folder. To enable or disable different device tree blobs, the `/boot/uEnv.txt` script, which defines the boot configuration, needs to be edited. A sample snippet of what uEnv looks like in default form can be seen below:

```
###U-Boot Overlays###
###Master Enable
enable_uboot_overlays=1
...
###Additional custom capes
#uboot_overlay_addr0=/lib/firmware/<file0>.dtbo
#uboot_overlay_addr1=/lib/firmware/<file1>.dtbo
...
###Disable auto loading of virtual capes (emmc/video/wireless/adc)
#disable_uboot_overlay_emmc=1
#disable_uboot_overlay_video=1
#disable_uboot_overlay_audio=1
#disable_uboot_overlay_wireless=1
#disable_uboot_overlay_adc=1
```

By default, there are no extra overlays loaded. In this project, the support for two SPI devices is needed, as well as two UARTs. To enable these interfaces, it is possible to simply edit the lines in the "Additional custom capes" section and add the paths of the corresponding device tree blobs located in `/lib/firmware/`. SPI0 and SPI1 were enabled, as well as UART1 and UART4. UART2's pins collide with SPI0 and UART3 is TX-only, which is the reason behind going all the way to UART4.

To avoid pin mux collisions, it is advised to remove the overlays for HDMI and multi-channel audio. This is done by enabling (uncommenting) `disable_uboot_overlay_video=1` and `disable_uboot_overlay_audio=1`. The ability to connect an external monitor or speakers is lost this way, but those were never going to be used in this project anyway.

Table 4.1 shows a diagram of the BeagleBone Black's P9 header and how its pins are mapped to various interfaces that were enabled and mentioned in this section. Please note that pins corresponding to greyed-out fields are not used, as well as pins with numbers 33 to 46.

BeagleBone Black's P9 Header			
Usage	**Pin#**		**Usage**
(FRAM) GND	**1**	**2**	
(FRAM) VDD (3.3 V)	**3**	**4**	
	5	**6**	
	7	**8**	
	9	**10**	
UART4_RX	**11**	**12**	
UART4_TX	**13**	**14**	
	15	**16**	
(FRAM) SPI0_CS0	**17**	**18**	SPI0_MOSI (FRAM)
	19	**20**	
(FRAM) SPI0_MISO	**21**	**22**	SPI0_SCLK (FRAM)
	23	**24**	UART1_TX
	25	**26**	UART1_RX
	27	**28**	SPI1_CS0
SPI1_MISO	**29**	**30**	SPI1_MOSI
SPI1_SCLK	**31**	**32**	

Table 4.1: A mapping of GPIO pins to interfaces enabled in this project

As mentioned in 2.4.4, the BeagleBone Black board used in this prototype includes a four gigabyte eMMC card used as a storage device. It is typically used to house the operating system and root filesystem, with the microSD card slot being used as expandable storage or a temporary boot device. As the eMMC storage is connected to the AM3358 processor in the same way as any other interfaces on this board, it too can cause pin collisions.

In the case of a pin collision being unavoidable, it is possible to disable the eMMC storage device completely. This means that only the microSD card can be used for booting, unless a new storage device is connected (e.g. via SPI). The `disable_uboot_overlay_emmc=1` line in uEnv.txt controls this behavior.

4.3 Porting of Filesystem Functionality

As mentioned earlier, the original iQtec PLC used the FatFs module to handle file reads and writes as well as other filesystem manipulation, such as creating, renaming and deleting of files and directories and accessing external storage. There were at least three ways of supporting this functionality going forward: either port FatFs to the new hardware and software configuration, use a different filesystem interface such as Boost.Filesystem or modules from the POCO project, or implement a new custom interface.

The decision to abandon FatFs and not use it in this project was made rather quickly. As mentioned in section 3.2, it offers sub-par performance and relatively low resistance to power loss events. Both Boost.Filesystem and the Foundation package from the POCO project, which includes filesystem functionality [38], were also looked into, but ultimately rejected. This project uses the C programming language strictly and both Boost and POCO are C++ libraries.

Instead, the decision was made to reimplement all functionality provided by FatFs from scratch using only standard C libraries. This task was quite straightforward, though it did have some interesting elements. The iQtec codebase is rather extensive and filesystem/file manipulation functions are used throughout the whole project in different contexts. In order to avoid hunting down each occurrence inside the codebase, it was decided to implement a filesystem handler that would emulate FatFs's application interface. This meant that the rest of the codebase could remain untouched.

Staying within the constraints of FatFs's API brought about some issues. Typically, filesystems use some sort of object information structure which contains a file's name, size, a date of creation and/or last modification, and information about its attributes (read-only flag, directory flag, etc.). This is also true in FatFs, its object information structure is named `FILINFO`. This structure is used when traversing directories and listing files inside of them.

```
typedef struct {
    FSIZE_t fsize;              /* File size */
    WORD    fdate;              /* Last modified date */
    WORD    ftime;              /* Last modified time */
    BYTE    fattrib;            /* Attribute */
    TCHAR   fname[12 + 1];      /* Object name */
} FILINFO;
```

An obvious solution to this is to just alias occurrences of `FILINFO` inside the codebase to a similar structure within the C libraries, but there is no one structure that contains all of the needed information. A new custom structure containing parts of both `stat` and `dirent` structures was devised. Using just `stat` would almost be enough, but there is a name field missing.

```
typedef struct {
    struct stat    stat_info;    /* size, date, time, attributes */
    struct dirent *dir_entry;    /* name */
} COMPLEXFILEINFO;
```

Another detail that needed straightening out was the logic behind creating and opening files in FatFs. The traditional `fopen` function which is a part of the C standard library and FatFs's `f_open` do not offer the same sets of options, which in some occasions called for some interesting workarounds. For example, when opening a file for *reading and writing in binary mode*, the FatFs implementation used the flags `FA_OPEN_ALWAYS | FA_READ | FA_WRITE`, which cannot be emulated in a single call of `fopen`. In some circumstances, up to 3 calls to `fopen` can happen in this new implementation. Though this might hamper the performance somewhat and put extra load on the storage device, this version of the filesystem handler is still orders of magnitude faster than FatFs, as discussed in 3.2.2.

Other than these two issues, the rest of the filesystem porting was completed without significant hitches.

4.4 SPI FRAM Driver, EEPROM Emulation

iQtec's original microcontroller-based PLC had multiple memory modules connected to it, with each serving a different purpose. A ferroelectric nonvolatile RAM module was connected via a serial peripheral interface. The FRAM's purpose was to store data from drivers communicating with peripheral data-gathering devices connected to it. A memory module was needed which would have a very long lifetime, as it typically incurred massive numbers of reads and writes - hence the usage of a ferroelectric RAM over an EEPROM or flash storage device.

Another memory module connected to the PLC was an EEPROM memory chip. This device was used for storage of two distinct types of data. The first type was logging information about devices connected to the PLC once their initialization was complete. This data was prepared and stored after boot-up and then never again. The amount of memory writes was dependent on the configuration of the whole home automation system, but the chip's endurance was not an issue even with a large amount of connected devices, as the system does not typically get rebooted often. The second use of the EEPROM chip was to store error logs. These errors were usually caused when a fault in communication with a peripheral device was detected, or when the peripheral device would stop communicating completely, often from a sudden loss of power. Again, the usage of an EEPROM chip is just fine in this usecase, as errors did not happen very frequently.

A handler function was a part of the original codebase that serviced requests for reads and writes to both the FRAM and EEPROM. The application decided which memory was to be accessed from the address and flag supplied to the function.

The decision was made to eliminate the EEPROM module from this project completely. Instead, information meant to be stored in the EEPROM would be redirected into two separate files located within the microSD card's filesystem. This helps simplify the system's architecture and bring a slight cost saving, as there is no more need to source and install the EEPROM hardware. The FRAM module functionality needs to be kept, as the frequency of accesses and its advantages are unchanged.

4.4.1 FRAM Transactions Over SPI

The FRAM device here is a Lapis MR45V100A unit, as described in 2.4.6. It is connected to the BeagleBone in a similar manner to the original PLC's design - via a serial peripheral interface (SPI), though only being connected to the GPIO pins and not to be CPU directly for prototyping purposes. The SPI_0 interface was chosen for this purpose. An illustration of how the GPIO pins are utilized here can be seen back in table 4.1. The SPI bus is usually enabled by simply loading the appropriate device tree overlay, e.g. `BB-SPIDEV0-00A0.dtbo`. In this case, the device tree had to be slightly modified. This will be explained in detail later.

A software interface was needed to be developed to integrate the FRAM's functionality into the rest of the system.

The *spidev* standard Linux userspace driver was used in the implementation. To initialize the bus device for use, first the appropriate `spidev` file (i.e. `/dev/spidev1.0` when using SPI_0 with the SPI_0_CS0 pin for chip select signaling) has to be opened and some of its parameters, such as read/write frequency or bits per word, configured. Once that is done, communication with the device can begin.

The `ioctl` system call is the core function used in all communication. Sending and receiving data is done using SPI transfers. An `spi_ioc_transfer` structure is created and its parameters are set; mainly the send and receive data buffers, message length and transfer rate (speed). `ioctl` is then called to dispatch the transfer message to SPIDEV.

FRAM reads and writes are implemented in a fairly conventional way - the TX array is first populated with command instructions and an address of where to store data / read data from in the FRAM module, and the rest of the transfer message is either blank if reading, or populated with actual data values if writing. After the transfer is completed, the RX transfer contains the expected values (nothing if writing or the contents of the FRAM chip if reading).

4.4.2 Setting Maximum SPI Speed in Device Tree

When an early version of the FRAM handler was implemented, the maximum communication speed (frequency) was set to be either the maximum speed of the FRAM chip itself, which is 34 MHz [6], or the maximum reported by SPIDEV by sending a transaction via `iotcl` with `SPI_IOC_RD_MAX_SPEED_HZ` as the argument, whichever of these two was lower. Everything seemed to be working just fine, but an issue was discovered where the transactions would seem subjectively a little too slow. The maximum theoretical throughput of the FRAM chip is around 30 Mb/s - it should be possible to overwrite the entire memory's contents multiple times a second. However after a test was conducted, the actual throughput was only around 3.2 Mb/s. After investigating, it was discovered that:

1. The maximum speed of SPIDEV0 is defined as only 16 MHz in the corresponding device tree overlay blob, even though the AM3358 should be capable of communicating at up to 48 MHz. [37]
2. `SPI_IOC_RD_MAX_SPEED_HZ` reports the maximum speed as 4 MHz every time.

A solution to problem number 1 was quite straightforward. First, the SPIDEV device tree overlay mentioned above had to be decompiled using the device tree compiler (DTC):

```
root@beaglebone:/lib/firmware# dtc -I dtb -O dts BB-SPIDEV0-00A0.dtbo \
> > BB-SPIDEV0-00A0.dts
```

Then, the maximum SPI speed field was changed to a more reasonable value (48 MHz):

```
...
__overlay__ {
    #address-cells = <0x1>;
    #size-cells = <0x0>;
    status = "okay";
    pinctrl-names = "default";
    pinctrl-0 = <0x1>;
    ti,pio-mode;

    channel@0 {
        #address-cells = <0x1>;
        #size-cells = <0x0>;
        compatible = "spidev";
        symlink = "spi/0.0";
        reg = <0x0>;
        spi-max-frequency = <0xf42400>;  <--- old value
        spi-max-frequency = <0x2dc6c00>; <--- new value
        spi-cpha;
    };
    ...
```

Note: 0xf42400 = 16,000,000, 0x2dc6c00 = 48,000,000.

Finally, the overlay was then recompiled again and deployed. The new device tree blob was renamed to reflect the modified SPI speed (by appending 48 to the old filename).

```
root@beaglebone:/lib/firmware# dtc -O dtb -o BB-SPIDEV0-00A48.dtbo \
> -b 0 -@ BB-SPIDEV0-00A0.dts
```

This solved the first issue.

The cause of the second problem, where the maximum reported SPI speed by the SPIDEV module was 4 MHz, was never discovered. The code was rewritten to set SPI speed to 34 MHz (the maximum fo the FRAM chip) in every transaction being processed, regardless of the reported maximum speed. This approach fixed the rather slow transaction speeds.

4.4.3 Real-World Transfer Speeds Before and After

Before applying these fixes, the performance of the FRAM chip was rather poor, only reaching write and read speeds of about 405 kiB/s. The throughput did not change at all even when the addresses and transaction lengths were randomized, indicating that the hardware was not in fact the bottleneck. After changing the device tree overlay and rewriting the handler to essentially hardcode the SPI speed before every transaction, the maximum read/write speeds reached 1600 kiB/s, or about 12.8 Mb/s of throughput. Though this is still lower than the theoretical maximum of 30 Mb/s, these speeds are satisfactory for now.

4.4.4 SPIDEV Maximum Buffer Sizes

During development and testing of the FRAM interface, an issue was encountered where relatively large read and write transactions would return a "Message too long" error when being processed. This was caused by creating transactions whose RX and TX buffers' length exceeded the maximum buffer size of SPIDEV, which is 4096 bytes by default.

There are two solutions for this issue - the default maximum length of the buffers can be overridden by either unloading the SPIDEV module and then reloading it with the `bufsiz=X` argument (using `modprobe`), or the uEnv.txt boot-up script can be modified by adding `spidev.bufsiz=X` to the `cmdline=` command, where X is the new maximum buffer size (see below). A third alternative solution is to implement the SPI FRAM handler to parse long transactions into multiple transactions with shorter message lengths.

```
cmdline=coherent_pool=1M net.ifnames=0 quiet spidev.bufsiz=16384
```

All three of the aforementioned solutions were tried and tested. While transaction splitting brings a tiny bit of extra overhead into the code, it guarantees portability. If the FRAM handling code ever gets reused on a different system, it should work just fine with no modifications needed to the SPIDEV module. Modifying uEnv was by far the simplest, but the uEnv file is subject to frequent changes and the `cmdline` command could be forgotten about when upgrading to a different distribution of the operating system. The one remaining approach - reloading SPIDEV with different arguments - was abandoned completely. In the end, the transaction splitting approach was deployed. The uEnv modification is part of the documentation, though unused at this time.

4.4.5 EEPROM Redirecting

As mentioned above, the original iQtec PLC used an EEPROM module to store error and logging information. In order to reduce costs and complexity slightly, it was decided to abandon the EEPROM completely. Instead, information originally meant for it would be stored in files on the microSD card.

Implementing this functionality was very straightforward. If no log and error storage files are present on the microSD card's filesystem, the application will first create empty binary files of a given size. Read and write requests meant to work with this data are simply pointed to these files, which are accessed using functions from the C standard library.

In order for this approach to work reliably, it is critical for the SD card to be mounted correctly. A script running at boot-up is set up to mount the SD card device as an external storage unit and check the integrity of these files.

In the original project, both the FRAM and EEPROM functionalities were implemented in a single set of handler functions. The handlers decided where to store data based on the flag supplied to the read/write function. As with the filesystem port, the reimplementation of this functionality was limited by the constraints of the original system's API. The original implementation included a lot of supplemental information and function calls because of its microcontroller-esque nature, such as low level hardware initialization functions. Pin IDs were also originally passed as arguments to most of the read/write request functions.

These function calls had to be kept in the ported version to mirror the API completely, though they did not need to offer any functionality - the ported version is purely in userspace and uses Linux drivers as a building block. Many of them were simply rewritten to do nothing.

An activity diagram of a write transaction request can be seen in figure 4.2.

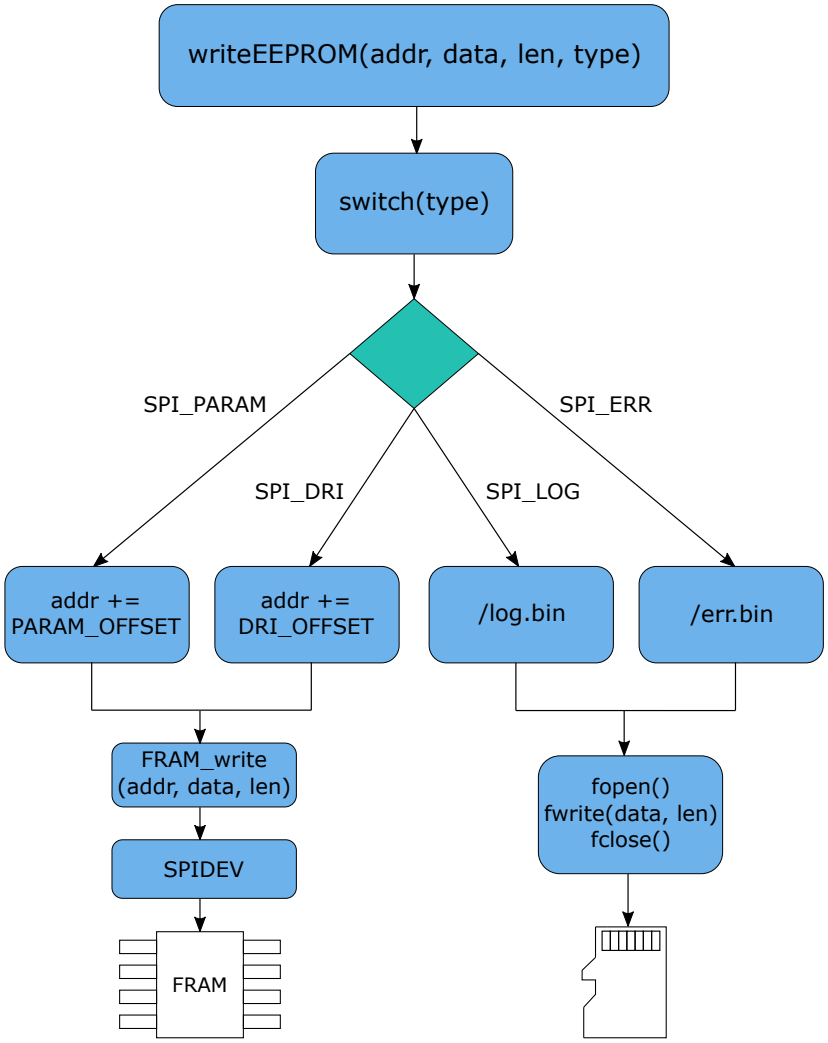

Figure 4.2: Activity diagram of ported FRAM/EEPROM handler

4.5 Userspace-Based RTC Driver

As mentioned in 2.1, there is an external real-time clock module or RTC installed as part of the original design of the PLC. It was decided to not use an external RTC in this project yet. One reason for this was that there is already an internal RTC integrated into the AM3358 processor.

Other than the rather obvious use of remembering time when the PLC is off, the RTC module is used in the iQtec project to keep precise time for tasks that require it. One of these is the filesystem - as there is no traditional clock in the original system, FatFs essentially didn't know what dates and times to put in a file's properties. Accesses to the RTC had to be implemented when originally porting FatFs to work on FreeRTOS.

The RTC naturally had to be accessed when the correct time had to be set as well. The PLC is set up to receive messages from either a UDP connection, via its integrated GSM modem, or via a file on the external storage. When a message is received from either of these three sources, it is parsed and the RTC is set accordingly.

Since this project uses a traditional operating system in which the current time is obviously a normal element, the argument could be made to not touch the internal RTC at all and instead just work with the system clock. Using the internal RTC provided a sense of reliability and additional precision and accuracy - which is not false, as the system clock is a software clock and as such is handled and maintained by the kernel. There is another upside in the fact that the AM3358's RTC is supported by standard Linux drivers. In the end, the RTC functionality was indeed ported to use the RTC instead of just system time.

To implement a port of the previous iQtec RTC functionality, first a file descriptor had to be connected to the device driver via typical open (and close) calls. The file for the internal RTC is usually located in `/dev/rtc0`. `ioctl` was used to gain access to the driver and dispatch transactions in a similar manner to the SPI driver described in section 4.4.1.

A typical function set had to be implemented in order for the port to work correctly; a function for reading the current time of day and date, another for converting between different time structure conventions. Another "standard" function returned the system's tick count (time elapsed since the boot-up of the system in milliseconds). All of this functionality is fairly traditional.

The biggest challenge of porting the real-time clock functionality was rooting through the codebase and writing code that would conform to the original iQtec API.

4.5.1 Setting Hardware Time

It is also important to note the userspace driver's behavior when attempting to write data to the RTC (e.g. calling `ioctl` with `RTC_SET_TIME`). It is recommended in the Linux documentation [39] to run the RTC-accessing program with the "set system clock; set real-time clock" (`CAP_SYS_TIME`) capability. An alternative is to simply launch the application with administrator privileges, i.e. `sudo`.

4.6 Accurate Software Timers

The iQtec home automation system can get very complex as the number of modules integrated within the system rises. Some modules depend on data provided from the PLC, some modules collect data and supply it to the PLC, some are a part of the PLC itself.

To function correctly, all modules and their functionality have to be served periodically. There can be dozens of tasks waiting for execution at any given moment during the system's operation, from data values waiting to be collected to communication requests that need to be answered. In order to process these tasks efficiently, a control loop is used and tasks are dispatched in a round-robin manner with predefined repeat times and priorities.

Tasks are usually organized in a queue and dispatched in sets. Each different task set is dispatched every 25 milliseconds, and a given set is scheduled once 40 ticks. This means that a given task is being handled once every second. There are also tasks that need to be dispatched much more frequently. These "high priority" tasks, such as the filesystem handler which processes read and write requests to the SD card (described in 3.2.2), are simply dispatched inside every task set, i.e. once every 25 milliseconds.

Some sort of a timer is needed to handle task dispatching as described above. Reasonably high precision is needed for this timer to make sure the frequency

of the tasks does not skew in time. The original project's microcontroller-based design used a hardware timer contained in its STM32F4 processor. This timer was configured to tick with a 1 millisecond period. An interrupt handler function for this timer decremented the "remaining wait time" value for each task in a queue. If the remaining wait time of a task reached zero, it was ready to be dispatched. The wait time of a given task reflected the "priority" (high or low) of it.

4.6.1 Proposed Solution: AM3358's PRU

Section 2.4.5 briefly mentioned a specialty of the Sitara AM3358 - in addition to the Cortex-A8 core, it also contains two Programmable Realtime Units, or PRUs for short. These chips are extremely simple, run at a clock rate of 200 MHz and are a great solution for usecases where very tight timings are required. The PRUs also have the ability to access the AM3358's RAM, making inter-chip communication possible.

At a glance, using one of the PRUs to implement an ultra-accurate timer would be possible. Some research was done on this topic, however it was decided in the end against pursuing this approach. One factor in the decision is the fact that the PRUs are programmed using a special PRU Assembly language, and thus have a very steep learning curve, which would compromise the time frame of this project. Code running on the PRU is also quite difficult to debug.

Another factor that played a role in the decision is the fact that the PRUs cause pin collisions with other interfaces used in this project (pin collisions and device tree overlays were explained in 4.2). It is essentially impossible to use the PRU subsystem in the current hardware configuration.

4.6.2 Software Approach

In 3.1.7, a benchmarking program was created to test timer-settings overheads. It was purposely designed with the reimplementation of task queue handling in mind. Most of the code from this benchmarking application was supposed to be reused here; first, a timer is set that fires every 25 milliseconds. After firing, a handler function is called whose only purpose is to raise a semaphore, set the timer again and exit.

The semaphore would be waited on by a completely different thread which is set up to be real-time and with the highest possible priority (99). This is where the actual processing of the task queue would be handled.

4.6.3 Minor Redesign

After starting the porting process, the structure had to be changed up slightly. As mentioned above, the original system deploys a 1 millisecond timer on which the task queue handling is dependent. There are also other unrelated constructs depending on this timer, so it cannot simply be abandoned.

In the actual reimplementation used in this project, the timer had to be reconfigured to fire with a period of 1 millisecond. The handler semaphore would only be raised once for every 25 tries, thus simulating a 25 millisecond timer. Another handler thread was added whose sole purpose was to handle "higher priority" or lower period tasks, as explained a few paragraphs ago. A different semaphore was used for this thread to wait on as well. This way, the higher priority tasks would essentially get their own queue and handler.

4.6.4 Specifics

The specifics of the reimplementation included the following concepts:

- A total of two POSIX-standard threads - one to handle the "fast" queue, the other to handle the regular queue
- Two POSIX semaphores, named `handler_sem` and `handler_sem_fast`
- A `pthread_attr_t` structure to preserve the real-time attributes of the two handler threads
- A `timer_t` structure to hold information about the main timer

All of these items (and more) were encapsulated in a single custom structure to keep clutter to a minimum.

The rather standard `timer_create`, `timer_settime`, etc. functions were used to implement this feature. The `SIG_ALRM` signal is fired when the timer

expires. The signal handler was designed to be very simple for multiple reasons. According to the documentation, a signal handler function should not ever call async-unsafe functions, or any standard library functions for that matter [40]. It is recommended that the signal handler only sets a flag that is processed by non-interrupt code, such as an entirely different thread as designed here. An upside of using an unrelated thread is that if a task set took too long to dispatch occasionally, the handler thread would deal with the delay and the timer would not run late, which would set the whole system back an undetermined amount of time.

The final structure of the software timer design can be seen in figure 4.3.

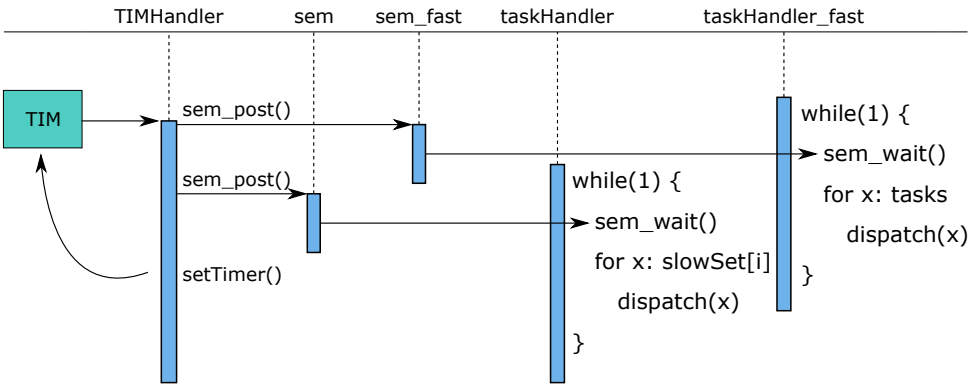

Figure 4.3: Diagram of the implemented software timer structure

In conclusion, even though this functionality is handled purely by software, which could provide sense of bad precision and delays, it proved to be quite reliable and returned very reasonable overhead times, even when the system is under high load. Using the PRU integrated in the AM3358 would definitely be very interesting and possibly prove to be an even better solution, but the implementation presented here is more than satisfactory. It also had the bonus of being somewhat programmer-friendly, a trait that is unsure in the PRU approach.

4.7 UART Interface Wrapper

The last part of the project that needed porting from the old iQtec system to the new Linux-based prototype was UART functionality.

In the original design, there are typically at least 4 UART links being used by the PLC, with the ability to use even more if necessary (e.g. if there is a demand for more nodes in the system; up to 16 UART links total). UART links are used to communicate both with devices which are a part of the PLC itself, and devices that are connected externally.

4.7.1 Uses

A device that is present in every PLC and is connected internally is the GSM modem. It is used for sending out data that is being collected from other modules within the automation system, as well as receive incoming connections. The GSM modem typically sees heavy use in a system where there is no accessible network being provided in the environment, be it for security or other reasons.

Other UART connections are used to communicate with modules that are typically in relatively remote locations - dozens or hundreds of meters away. To ensure a reliable connection over longer distances, the RS-422 and RS-485 communication standards are used via standard UART-RS-4xx converters. Another mode of communication used here and connected via UART is an M-Bus interface. M-Bus (or Meter Bus) is typically used to communicate with electric power, gas or water meters in order to get readouts from those devices. The usecase of M-Bus in this project is quite obvious - a home automation system is going to want to read data from power or gas meters for various purposes.

The original project's UART handlers are quite low-level because of its microcontroller nature. It is not possible to reuse the same code as there is no readily-available hardware accessibility. Instead, the UART functionality is again ported using Linux drivers.

It was decided to implement only some very basic UART handler functionality for now - functions that would setup a UART device with given parameters, functions that handled sending a receiving data and some basic error detecting code (CRC) calculation.

4.7.2 Implementation Specifics

C's `termios` library functions were used to implement the majority of the above functionality. A structure was created that takes care of all UART communicators by their ID. A total of three UARTs (two are mentioned in 4.2, plus the UART0 serial console) are enabled in the prototype for now. There are files in the `/dev/` directory corresponding to their respective interfaces (`/dev/ttyO1`, `/dev/ttyO4`). To initialize a given serial device, its corresponding file needs to be opened and the communication's properties have to be configured using the `termios` structure and `tcsetattr` function.

```
...
snprintf(file, sizeof file, "/dev/ttyO%d", devNum);
serial_fd = open(file, O_RDWR | O_NOCTTY | O_NDELAY);
struct termios config;
memset(&config, 0, sizeof(config));
config.c_iflag &= ~(IGNBRK | BRKINT | ICRNL | INLCR |
                    PARMRK | INPCK | ISTRIP | IXON);
config.c_oflag = 0;
config.c_lflag &= ~(ECHO | ECHONL | ICANON | IEXTEN | ISIG);
config.c_cflag &= ~(CSIZE | PARENB);
config.c_cflag |= CS8;
cfsetospeed(&config, baudrate);
tcsetattr(serial_fd, TCSANOW, &config);
...
```

There is a total of four different flag fields in the `termios` structure (in order of appearance above) [41]:

- Input modes - low-level aspects of input processing - line ending translation types, handling of framing or parity errors, input data flow control
- Output modes - modifying and padding of output data - line ending translation, tab translation (converts tabs to spaces)
- Control modes - parameters usually associated with asynchronous serial data transmission - enable/disable/set types of parity used, use local connection (no modem)
- Local modes - higher-level aspects of input processing - echoing, signals

In this implementation, accesses to the serial ports are non-blocking. To make sure data is written or read out of the device correctly, a wrapper was created with input and output buffers, and a periodical flushing of both buffers was implemented. If the device is not ready or has no data to be read out, the `EAGAIN` error is received and the flushing function simply does nothing and tries again next time. Each serial device has its own set of input and output buffers and wrappers.

As was the norm when porting other functionalities described in this chapter so far, the constraints of the original iQtec system's API had to be adhered to. This meant that functions for reading and writing of single bytes, as well as longer character strings had to implemented.

A cyclic redundancy check function was implemented as well, which is used to provide error checking of inbound and outbound messages and commands. The algorithm for CRC calculation and verification was simply carried over from the original project's codebase, as was an implementation of the serial framing protocol. These were left largely untouched and worked just fine when combined with the other ported functionality.

Since the original UART code was rather low-level, most of the initialization functions were either kept empty or very sparse, as there was no need to setup interrupts or clocks. A big drawback here is the amount of available UART links - the hardware in its current configuration does not allow for more than three at this moment. Pin multiplexing will have to be examined and device trees will need changing around to allow for more UARTs in the future.

4.8 Standalone Application Substitutes

4.8.1 FTP Server

The iQtec PLC also includes the ability to act as a simple FTP server. This functionality is typically used to push new firmware onto the device remotely or retrieve log and error information or other data files from the integrated microSD card. There is a separate thread dedicated to serving FTP functionality in the original implementation.

4.8.2 FTP's Security Issues

The old FTP server implementation was discarded when developing this project. While it would be possible to simply port the code over to the new project, a decision was made to instead install and use a standalone application.

A huge factor in this decision is the fact that the original file transfer protocol has some glaring security issues. When logging onto the server, the login information (username and password) are being sent as plain text with no sort of security measures applied. The data being transmitted is also completely plain with no kind of encryption, making it vulnerable to being intercepted. This is a very non-secure behavior and could easily be exploited by a person with malicious intents.

There are multiple different alternatives to using FTP that provide various degrees of improved security. FTPS is an extension of the FTP protocol that adds in-transit encryption using SSL (secure socket layer), making it essentially "FTP over SSL." [42] Packet interception is still possible, though the data inside would not be of any use to an attacker. It is still possible to use an unencrypted control channel with FTPS (encryption of the control channel is not strictly required), which is not ideal.

Another alternative protocol is SFTP, which stands for "SSH file transfer protocol." SFTP does not provide the option to use an unencrypted control or data channel, which means that by default it does not broadcast any information in plain text. Another good thing about SFTP is the fact that,

in a similar fashion to FTP, it only requires a single port to be open for both control and data streams, making it friendly to client-side firewalls. [42]

An SFTP server was installed using Debian's APT package manager and configured to accept the same sort of credentials as the old FTP system. The original implementation was compliant to FTP standards, i.e. it supported all standard commands and used the proper return codes. This means that the new solution works just fine out of the box.

4.8.3 HTTP Server

The original design also included an HTTP server which could serve a simple web interface used for both monitoring and configuring a given iQtec automation system. Originally, this functionality was based on *lwIP* - the Lightweight TCP/IP stack, which was used for its relatively low demand for system resources, namely memory. The HTTP server offered no security whatsoever, though.

Again, there are multiple alternatives that could be used in place of the original solution. One of them was the inclusion of `microhttpd`, which is a small C library that could be used to replace the lwIP functionality in the previous system design.

A better idea is to employ the same approach as described in the previous section - don't integrate this functionality at all and instead use a standalone application. There are dozens of Linux HTTP servers that provide HTTPS - all that remains is to make a choice. The HTTP server functionality has not yet been re-added.

Chapter 5

Conclusion

The goal of this thesis was to research the options of migrating a rather complex microcontroller-based home automation system to a completely new design and develop a prototype that could provide the same functionality.

In chapter one, the existing market for these home automation systems is overviewed, the historical progression of this field of computing and its current trends are explained, and some basic concepts are discussed.

The second chapter first provides a closer description of the existing iQtec home automation system's hardware and system requirements are established and discussed for its replacement. Next, some replacement candidates are introduced and their degrees of requirement satisfaction are discussed. Finally, the proposed hardware solution that was chosen is described in greater detail and its specifics are introduced.

Chapter three provides a detailed explanation and comparison of the old and new systems' software solutions. This includes a description of the inner workings of real-time operating systems and an explanation of some key concepts related to real-time computing. The PREEMPT_RT patch for Linux kernels is described and its effects are measured. Weak points of the existing home automation system are identified and alternative solutions are explained. The software development tools and solutions used in creating the new system's prototype are also explained.

Chapter four describes in detail all of the tasks and efforts expended in order to port, reimplement and otherwise emulate the old system's functionality using approaches described in chapter three. In addition, smaller details are explained in individual sections. The Linux device tree system is introduced and some of its specifics are examined. The process of implementing a filesystem handler, SPI-connected ferroelectric RAM functions, integrated RTC controls, a wrapper for serial communication and software timer emulation functionality is described, issues during development are explained and their solutions are introduced. Standalone application solutions for some smaller parts of the original project are also explored.

The research portion of this project had the goal of selecting a hardware solution that would be suitable to the task of executing the project. An Octavo OSD3358 System-In-Package was chosen at first, but was substituted by a Texas Instruments Sitara AM3358 early on in the development stage of the project. This had next to no effects on the system's functionality, as the OSD3358 does in fact incorporate the AM3358. Once a different Ethernet device gets sourced and integrated into the system, a switch can happen to the original plan of using the OSD3358.

Section 2.3.1 defined some important hardware system requirements which were derived from research conducted on the old iQtec system and an analysis of its weak points. I believe the choice of hardware solution for this project in accordance with these requirements was done correctly. The Octavo OSD3358/AM3358 does indeed satisfy all but one of the defined hardware system requirements, though its absence of multiple cores did not prove to be an issue. It offers good performance when compared to the other proposed solutions, and is far more powerful than the previous solution. The inclusion of an eMMC storage device is also a great feature. Once the issue of a relatively low amount of UARTs being available in the current system configuration gets rectified, the OSD3358 can potentially be considered a perfect fit for use in a system like the one prototyped in this project. There were no other significant issues that would suggest otherwise.

I believe the design and software development of this project's prototype was a success. Weak points identified during the research phase and explained in chapter three were successfully eliminated or mitigated with tests and measurements to support this claim; significant performance gains were observed across the board. The tasks of porting or reimplementing critical functionality over to the prototype were accomplished.

Not all approaches planned for execution were pursued, such as the usage of the Sitara AM3358's Programmable Real-time Units for precise timer interrupts. Other issues were identified that will need more effort to solve, such as the relatively low amount of available UART links.

This project showed that it is indeed viable to migrate the iQtec home automation system from its current hardware and software solutions to solutions described in this thesis, though more work needs to be done to complete this task in its entirety. A solid foundation has been laid down upon which more development work can be conducted and integrated.

5.1 Future Development

In the future, more development will need to be done in order to complete the migration of the iQtec home automation system to the new Linux-based approach.

Important tasks which need more exploration include buttoning up of communication protocols briefly mentioned in chapter 4, such as RS-485 and RS-422, that facilitate communication with other nodes in the home automation system. Solving the low UART count problem is also an important future task. A good deal of other smaller, less important tasks is also available for solving, such as the HTTP server functionality mentioned at the end of chapter 4.

Bibliography

[1] Real Time Engineers Ltd.: *About FreeRTOS; RTOS - Free professionally developed and robust real time operating system...* [online]. 2019, [visited on 2019-04-14]. Available from: `https://www.freertos.org/RTOS.html`

[2] Knud Lasse Lueth: *State of the IoT 2018: Number of IoT devices now at 7B – Market accelerating* [online]. 2018, [visited on 2019-05-10]. Available from: `https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/`

[3] Ericsson: *Ericsson Mobility Report June 2018* [online]. 2018, [visited on 2019-05-10]. Available from: `https://www.ericsson.com/assets/local/mobility-report/documents/2018/ericsson-mobility-report-june-2018.pdf`

[4] IEEE: *The International Roadmap For Devices and Systems* [online]. 2017, [visited on 2019-05-10]. Available from: `https://irds.ieee.org/images/files/pdf/2017/2017IRDS_MM.pdf`

[5] iQtec.cz: *iQtec Home Automation System*. Marketing material, 2014.

[6] Lapis Semiconductor: *MR45V100A* [online]. Datasheet, 2018, [visited on 2019-03-22]. Available from: `http://www.lapis-semi.com/en/data/datasheet-file_db/Memory/FEDR45V100A-02.pdf`

[7] Variscite: *DART-6UL Product Brief* [online]. 2019, [visited on 2019-05-07]. Available from: `https://www.variscite.com/wp-content/uploads/2017/12/DART-6UL_Product_Brief.pdf`

[8] STMicroelectronics: *STM32MP153C datasheet* [online]. Datasheet, 2019, [visited on 2019-05-07]. Available from: `https://www.st.com/resource/en/datasheet/stm32mp153c.pdf`

[9] STMicroelectronics: *STMicroelectronics Launches STM32MP1 Microprocessor Series with Linux Distribution to Speed IoT and Smart Industry Innovation* [online]. 2019, [visited on 2019-05-07]. Available from: `https://www.st.com/content/st_com/en/about/media-center/press-item.html/p4140.html`

[10] Octavo Systems: *OSD335x Family* [online]. Datasheet, 2019, [visited on 2019-02-15]. Available from: `https://octavosystems.com/docs/osd335x-datasheet/`

[11] BeagleBoard.org Foundation: *BeagleBoard.org - Black* [online]. 2019, [visited on 2019-03-02]. Available from: `https://beagleboard.org/black`

[12] Texas Instruments: *AM3358 Sitara Processor: Arm Cortex-A8, 3D Graphics, PRU-ICSS* [online]. 2011, [visited on 2019-02-26]. Available from: `http://www.ti.com/product/AM3358`

[13] Wikipedia: *FreeRTOS — Wikipedia, The Free Encyclopedia*. 2019, [visited on 2019-04-18]. Available from: `https://en.wikipedia.org/wiki/FreeRTOS#Implementation`

[14] Real Time Engineers Ltd.: *Why RTOS and What is RTOS?* [online]. 2019, [visited on 2019-04-15]. Available from: `https://www.freertos.org/about-RTOS.html`

[15] Ts'o, T.; Hart, D.; Kacur, J.: *What is real-time? - RTwiki* [online]. 2012, [visited on 2019-04-21]. Available from: `https://rt.wiki.kernel.org/index.php/Frequently_Asked_Questions#What_is_real-time.3F`

[16] The Linux Foundation: *Intro to Real-Time Linux for Embedded Developers* [online]. Blog article, 2013, [visited on 2019-04-20]. Available from: `https://www.linuxfoundation.org/blog/2013/03/intro-to-real-time-linux-for-embedded-developers/`

[17] The Linux Foundation: *Preemption Models* [online]. 2016, [visited on 2019-04-20]. Available from: `https://wiki.linuxfoundation.org/realtime/documentation/technical_basics/preemption_models`

[18] Cerqueira, F.; Brandenburg, B. B.: *A Comparison of Scheduling Latency in Linux, PREEMPT_RT, and LITMUS^{RT}* [online]. Proceedings of OSPERT 2013 9th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications, 2013, [visited on 2019-04-30]. Available from: `https://people.mpi-sws.org/~bbb/papers/pdf/ospert13.pdf`

[19] The Linux man-pages project: *sched(7)* [online]. 2019, [visited on 2019-04-30]. Available from: http://man7.org/linux/man-pages/man7/sched.7.html

[20] The Linux Foundation: *Cyclictest - FAQ* [online]. 2018, [visited on 2019-04-22]. Available from: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/faq

[21] Waterland, A.: *Stress project page* [online]. 2014, [visited on 2019-04-22]. Available from: https://people.seas.harvard.edu/~apw/stress/

[22] Clifford: *Running applications from freeRTOS* [online]. 2014, [visited on 2019-04-23]. Available from: https://stackoverflow.com/revisions/21921675/3

[23] Clifford: *What is the difference between RTOS and Embedded Linux?* [online]. 2014, [visited on 2019-04-23]. Available from: https://stackoverflow.com/revisions/25875777/2

[24] ChaN: *FatFs - Generic FAT Filesystem Module* [online]. 2019, [visited on 2019-04-22]. Available from: http://elm-chan.org/fsw/ff/00index_e.html

[25] STMicroelectronics: *Developing applications on STM32Cube™ with FatFs* [online]. User manual, 2019, [visited on 2019-04-26]. Available from: https://www.st.com/resource/en/user_manual/dm00105259.pdf

[26] Chen, R.: *Why is the FAT driver called FASTFAT? Why would anybody ever write SLOWFAT?* [online]. Microsoft Developer Blogs, 2014, [visited on 2019-04-27]. Available from: https://devblogs.microsoft.com/oldnewthing/20141030-00/?p=43733

[27] Munegowda, K.: *Power Fail Safe FAT File System* [online]. Embedded Linux Conference (Lecture), 2011, [visited on 2019-04-27]. Available from: https://elinux.org/images/5/54/Elc2011_munegowda.pdf

[28] Jones, M. T.: *Anatomy of Linux journaling file systems* [online]. IBM developerWorks, 2008, [visited on 2019-04-29]. Available from: https://www.ibm.com/developerworks/library/l-journaling-filesystems/l-journaling-filesystems-pdf.pdf

[29] The Kernel Development Community: *Directory Entries — The Linux Kernel documentation* [online]. 2019, [visited on 2019-04-29]. Available from: https://www.kernel.org/doc/html/latest/filesystems/ext4/directory.html#hash-tree-directories

[30] Linaro: *Toolchain - Linaro* [online]. 2014, [visited on 2019-04-30]. Available from: https://wiki.linaro.org/Toolchain

[31] Linaro: *Toolchain Frequently Asked Questions - Linaro* [online]. 2017, [visited on 2019-04-30]. Available from: https://wiki.linaro.org/WorkingGroups/ToolChain/FAQ

[32] Molloy, D.: *Exploring BeagleBone: Tools and Techniques for Building with Embedded Linux*. John Wiley & Sons, Inc., 2015, ISBN 978-1-118-93512-5, pages 219-221.

[33] Embedded Linux Wiki: *Device Tree Usage - eLinux.org* [online]. 2019, [visited on 2019-05-02]. Available from: `https://elinux.org/index.php?title=Device_Tree_Usage&oldid=491246`

[34] Eshtaartha Basu: *OSD335x Lesson: Linux Device Tree* [online]. 2018, [visited on 2019-04-27]. Available from: `https://octavosystems.com/app_notes/osd335x-design-tutorial/osd335x-lesson-2-minimal-linux-boot/linux-device-tree/`

[35] Embedded Linux Wiki: *Capemgr - eLinux.org* [online]. 2017, [visited on 2019-05-02]. Available from: `https://elinux.org/index.php?title=Capemgr&oldid=431081`

[36] Robert C. Nelson: *Where did the slots file go? - eLinux.org* [online]. 2019, [visited on 2019-05-02]. Available from: `https://elinux.org/index.php?title=Beagleboard:BeagleBoneBlack_Debian&oldid=490826#Where_did_the_slots_file_go.3F`

[37] Texas Instruments: *AM335x Sitara Processors datasheet* [online]. Datasheet, 2018, [visited on 2019-05-02]. Available from: `http://www.ti.com/lit/ds/symlink/am3358.pdf`

[38] Applied Informatics Software Engineering GmbH: *POCO C++ Libraries - About* [online]. 2019, [visited on 2019-05-02]. Available from: `https://pocoproject.org/about.html`

[39] The Linux man-pages project: *rtc(4)* [online]. 2017, [visited on 2019-04-17]. Available from: `http://man7.org/linux/man-pages/man4/rtc.4.html`

[40] ISO/IEC: *International Standard ISO/IEC 9899 (Programming languages - C)* [online]. Manual, 2011, section 7.1.4 [visited on 2019-05-06]. Available from: `http://www.iso-9899.info/n1570.html`

[41] Free Software Foundation, Inc.: *The GNU C library* [online]. Manual, 2008, [visited on 2019-05-08]. Available from: `https://www.gnu.org/software/libc/manual/html_node/Concept-Index.html`

[42] Horan, M.: *SFTP vs. FTP: Understanding the Difference* [online]. 2017, [visited on 2019-05-07]. Available from: `https://www.ftptoday.com/blog/sftp-vs.-ftp-understanding-the-difference`

[43] Siewert, S.; Pratt, J.: *Real-Time Embedded Components and Systems with Linux and RTOS*. Mercury Learning & Information, second edition, 2016, ISBN 978-1942270041.

[44] Abbott, D.: *Linux for Embedded and Real-time Applications (Embedded Technology)*. Newnes, third edition, 2012, ISBN 978-0124159969.

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Mrázek** Jméno: **Jan** Osobní číslo: **406809**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra měření**

Studijní program: **Otevřená informatika**

Studijní obor: **Počítačové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Migrace a reimplementace funkcí IoT systému reálného času

Název diplomové práce anglicky:

Migration of an IoT real-time system and reimplementation of some of its functions

Pokyny pro vypracování:

1) Nastudujte hardwarové a softwarové řešení systému inteligentního řízení domu firmy iQtec.
2) Porovnejte výhody, nevýhody a rozdíly hardwarového přípravku založeného na čipu Octavo OSD335x oproti dosavadnímu řešení.
3) Prostudujte problematiku týkající se použití operačních systémů reálného času.
4) Navrhněte a realizujte implementaci některých funkcí systému při použití tohoto řešení.

Seznam doporučené literatury:

[1] Sam Siewert, John Pratt: Real-Time Embedded Components and Systems with Linux and RTOS, 2016 Mercury Learning & Information. ISBN 978-1942270041.
[2] Doug Abbott: Linux for Embedded and Real-time Applications, 2012 Newnes. ISBN 978-0124159969.
[3] Firemní materiály, dokumentace firem Octavo, Texas Instruments

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Milan Kolář, Prologue s.r.o.

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **24.01.2019** Termín odevzdání diplomové práce: _____

Platnost zadání diplomové práce:
do konce letního semestru 2019/2020

Ing. Milan Kolář podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____ Datum převzetí zadání

_____ Podpis studenta