Flexiware Updates

Improving Efficiency of Firmware Updates for Resource-Limited Devices in a User-Friendy Manner

Camilla Kusk & Steffen Pedersen
Computer Engineering - Networks & Distributed Systems, 1049B, 2025

Master Thesis

This report is typeset using the Typst system.

AALBORG UNIVERSITY
STUDENT REPORT

Department of Electronic Systems
Aalborg University
https://www.es.aau.dk

Title:
Flexiware Updates

Theme:
Improving Efficiency of Firmware Updates for Resource-Limited Devices in a User-Friendy Manner

Project Period:
Spring Semester 2025

Project Group:
1049B

Participants:
Camilla Kusk
Steffen Pedersen

Supervisors:
Tatiana K. Madsen
Rasmus S. Frederiksen

Copies: 1

Number of Pages: 54

Date of Completion:
04/06-2025

Abstract:

The number of IoT devices continues to grow, including those deployed in remote, resource-constrained environments. Maintaining these devices requires regular firmware updates, which must be energy-efficient to preserve battery life. This project focuses on improving the firmware updates for low-energy devices by minimizing update file size through memory management and delta encoding techniques. The improvement is based on the assumption of a direct correlation between the file size and the energy spent on the update.
The proposed solution involves a Custom Linker Script that leverages dynamic linking to produce smaller, memory-aware update files. Several test scenarios were used to evaluate the effectiveness of the tool, with results showing reduced firmware sizes in most cases. While the usability of the tool remains an area for improvement, the findings demonstrate that memory-aware linking can significantly reduce update size and, by extension, energy consumption during firmware updates. Thus improving the lifetime operation of the devices.

Contents

Introduction 1

Internet of Things (IoT) devices continue to increase in the billions. There are approximately 18 billion devices currently, and in 2030, it is expected to reach 40 billion. These devices aid the modern world in everything from data collection to convenience. 80% of *IoT* devices use either *Wi-Fi* (31%), Bluetooth (25%), or Cellular (21%) [1]. *IoT* devices can be found in every field, from smart homes to industries such as the medical, infrastructure, agriculture, maritime fields, etc. An improvement in any aspect might be huge when considering the scale and variety of their use cases. [2]

The majority of *IoTs* are battery-powered, meaning that power consumption, or more precisely the *Energy Mangement (EM)*, focuses on the longevity of the devices by optimizing the sleep, sampling, and sending of data. By introducing sleep cycles, efficient data processing, and use communication protocols in an efficient manner, such that the individual devices' lifetime operation is considered. Some communication protocols like it have been made to be more efficient and reduce energy usage by adjusting the transmission parameters. Other methods of making the device more power efficient through communication to reduce overhead or lowering the transmission frequency. These energy-efficient *IoT* devices have both in terms of their hardware, software, and communication protocols been changed for the main purpose of reducing the total power consumption [3]. However, because of all the ways of improving efficiency, it reaches a point where each part's optimization might limit other aspects in terms of future changes.

To narrow the scope, remote environments where devices have limited bandwidth, such as satellites, where the constraints imposed cannot be changed easily or would be too expensive to mitigate. Given these challenges, it is crucial to minimize the time the device spends in its active state, such as during data processing or updates. Updating, or rather, firmware updates, which are low-level code that directly interacts with the hardware of the device, and ensures that it performs its functions correctly, are key aspects to the longevity of the devices. Regular firmware updates can help enhance security, enable new features, comply with newer industry standards and regulations, as well as help device performance and increase its lifespan as a result *[4]*.

1.1 Initial Problem Statement

To conserve limited resources and battery life, it is essential to minimize the device's active time during processes like data handling or firmware updates. In remote environments, these constraints make full firmware updates impractical.

How can firmware updates be optimized for devices with limited resources available?

Problem Analysis 2

Firstly, the different aspects of this problem will be identified and analyzed. Starting at a higher level, by finding out which devices are usually used in these kinds of use cases. From there, the different levels of performance and power usage is possible to narrow down and analyze.

2.1 Low Energy Devices

Most *IoT* sensors are placed either in conditions where the temperature is strictly monitored or to collect information about the area. With rapid growth and deployment of *IoT* devices across a wide range of fields, from agriculture to aerospace, have brought significant attention to their long-term maintainability, particularly in power-constrained and remote environments. As discussed in the chapter *1: Introduction*, these devices are often optimized to consume as little energy as possible, which poses a major challenge when trying to apply firmware updates that require extended active operation *[3]*.
Ideally, these low-energy devices must be very efficient at every step, such that their operational life is prolonged. From a generalized perspective, the states of the device can be divided into the three states:

- Wakeup state
- Processing state
- Standby state

A quick wakeup and processing state would be ideal while also balancing the usage with how much energy is spent on it, such that a sort of equilibrium is found between power and processing usage. Meanwhile, the standby state is mostly focused on maintaining low energy usage. Firmware updates, however, disrupt this balance. They require sustained processing, active communication, and storage writes all of which occur in the high-power states. Therefore, even a single firmware update can significantly impact the device's energy budget. Based on this, minimizing the time spent in the processing state should be a priority *[4]*.

2.1.1 The Need for Firmware Updates

Most of the embedded devices are placed such that they are meant to stay for a long time. So, it would be natural to assume that they should never need an update. What are the reasons firmware updates are necessary?

While many embedded *IoT* devices are designed for long-term deployment without intervention, firmware updates remain essential for several reasons. Regular firmware updates can mitigate security risks, system failures, and impact performance. Regular updates will

enhance the security, fix bugs, and could also introduce new features if not enhancing the existing ones. Lastly is compatibility, as the external devices that communicate with them can change, which might require a change in the embedded device as well *[5]*. In remote and resource-limited environments, applying these updates becomes particularly challenging, not only because of energy constraints but also due to bandwidth limitations and the risks associated with interrupting normal operation. This makes the need for efficient, reliable update mechanisms more important than ever.

2.2 Update Methods

The term update is very broad; as such, it can, in the context of this report, be subdivided into categories, which will provide an easy overview of the update process:

- **Update mechanism**
  - A/B updates
  - Rolling update
- **Resource use**
  - Full update
  - Delta update
  - Binary Patching
- **Update Execution Architecture**
  - Static Linking
  - *Positional Independt Coding (PIC)*
  - Dynamic Linking
  - Segmented Linking (Linking Segmented Code)
- **Distribution Channel**
  - *Over the Air (OTA)*
  - *Peer-to-Peer (P2P)* Update

Update Mechanism

These subcategories can be useful for identifying what approach to take when discussing future updates on embedded hardware. The first element regarding the update mechanism is the *A/B update*, which uses a safety mechanism by having two slots: “A” and “B”, with both slots containing an updated version. The slot with the newest update is the one running, while the other slot includes an older version, so if the new version is faulty, there is still something to fall back on. With A/B updates, the bootable dual partitioning system has the advantage of a fail-safe *[6]*.

Another update mechanism is the *Rolling updates*, depending on the ecosystem, this name might differ, as some refer to them as *phased* or *canary deployment* as a reference to the canary in the mines. This means that the update will be sent out in phases, so if something were to go wrong, it would only affect a certain percentage of the devices. Using this method, it is possible to test the reliability of a newly released update *[7]*.

Resource Use

Resource use refers to the size of the update that needs to be sent. However, this can be subdivided based on the specific method used to "make" the update. A full update is, as the name suggests, the complete code sent. A delta update only sends the changes between the updated version and the one currently running on the system. This makes delta updates more efficient than the full update, as only the necessary parts of the code are sent to the devices. Binary patching directly modifies the code without the need for source code and recompilation. It is often used for fixing bugs or implementing a small feature.

Update Execution Architecture

The update execution architecture highlights how the updates are linked, loaded, and executed, which plays a critical role in the system's flexibility and performance.
To properly understand *Dynamic Linking*, it helps to contrast it with *Static Linking*. Static linking combines all necessary libraries and dependencies into a single executable at compile time. This makes the program self-contained and portable, but often results in larger file sizes and longer build times due to full dependency inclusion.

Dynamic linking, by contrast, loads external libraries at runtime. This allows multiple programs to share the same library in memory, reducing both file size and update complexity, especially when only the libraries change. Dynamic linking also supports modularity, making it easier to update components without rebuilding the entire application *[8]*, *[9]*.
This is where *PIC* becomes essential as it allows code to be compiled and executed correctly regardless of its memory address, which is a key part of dynamic linking. Since shared libraries may be loaded at different addresses in different programs, *PIC* ensures they function correctly without recompilation. *PIC* is also useful in modular embedded systems where code must be loaded and relocated dynamically. The opposite of PIC is absolute code, which relies on hardcoded addresses and is much less flexible for such purposes *[10]*, *[11]*.

The flexibility provided by PIC also plays a crucial role in systems that use *segmented code linking*. In embedded systems, especially those with limited memory, code is often divided into segments or modules that can be updated or loaded independently. *Linking segmented code* involves resolving addresses across these separately compiled sections, which can be challenging if addresses are fixed. By using PIC, each segment becomes relocatable, making it easier to load modules into different memory locations without requiring changes to the binary files. When combined with dynamic linking, segmented code can be loaded, replaced, or updated on-the-fly, enabling modular firmware designs that support dynamic updates and reduce downtime in constrained devices *[12]*.

	Static Linking	Dynamic Linking
Definition	Everything is combined into a single file at compile time.	External libraries are referenced at runtime, when the program is loaded/executed.
File Size	Generally larger as a result of the combination.	Smaller size, because of the external libraries dynamically linked at runtime.
Flexibility	Less flexible because of larger file size, which means a long compile time if an update is needed, but more portable as a result of everything contained in a single file.	More flexible, as updates in the libraries can be done without the need for recompiling the program.
Performance	Faster program start and direct execution.	Slightly slower startup due to linking, but overall minimal impact on performance

Table 2.1: Summary Table of the main differences between Static and *Dynamic Linking [8]*.

Distribution Channel

The Distribution channel refers to how the update is sent to each device. *OTA* delivers the firmware update wirelessly via. networks such as *Wi-Fi* or cellular connections. This type of update is very common for devices that are located in remote or hard-to-reach locations. Some of the key benefits of using *OTA* are the scalability, as it facilitates updating large numbers of devices simultaneously. However, there are also considerations such as security risks involved and the reliance on a network connection. However, *OTA* updates can also enable a timely patching against vulnerabilities because of its key benefits *[13]*.

P2P updates take a decentralized approach, where the devices themselves share updates directly with each other instead of solely relying on a central server. The key advantage of this update type is its application in environments with limited connectivity. The *P2P* topology has an increased complexity as a result of its flexibility and architecture. This means that robust protocols for device discovery and communication as to not cause network congestion or too much uptime for the individual devices. As such, aside from security considerations, managing the version control for the devices is also essential *[14]*, *[15]*.

2.2.1 Scalability & Efficiency Tradeoff

Due to the constraints of these low-powered devices, there is likely a significant constraint on available storage on the device. This means that the update also needs to be considered in the context of the use case.
Making a device fully upgradable, with near infinite room for the firmware to expand into, could easily seem excessive, if there is a 99.9% certainty that all that will ever be needed to change is a couple of integer values, which will stay within the same size constraints anyway.

2.3 State of the Art

This section explores the current *State of the Art (SoA)*, which will give an insight into the methods currently applied and discuss the advantages and disadvantages of these methods. These methods might also need different metrics to evaluate their advantage, as such a table will summarize the key takeaways for this project.

In this first paper, *[16]*, Zephyr, an *Real-Time Operating System (RTOS)* ecosystem OS, is used to compare against other solutions, such as scripting-based approaches and virtual machines. However, it starts by categorizing the method for the update type into four distinct types:

- 1) *Non-Volatile Memory Flashing (Full, Partial)*
- 2) *Dynamic Loading*
- 3) *Virtual Machine*
- 4) *Scripting*

The first is also what has been referred to as an A/B update earlier in this report. Dynamic loading often refers to approaches where *PIC* is used. With a loadable code, it is possible to make changes without the need for a reboot, which does have the advantage of being very efficient; however, if poorly implemented can cause the software to crash. The third type, using *Virtual Machines (VMs)*, in these solutions, the performance takes toll as there is an inherent indirect memory management of the system. The fourth refers to the use of high-level languages (Python or JavaScript) that then convert the language of the code to either bytecode or machine code.

Their main solution is the **"Dynamic App Loading for Zephyr" (DAL)**, where they essentially create an *Software Development Kit (SDK)*. Zephyr lacks native dynamic code loading, but a community-developed solution, "DAL", enables it via kernel syscalls and threads. DAL isolates application code from the kernel, requiring custom syscalls for interaction. Apps are written in C, compiled as *PIC*, and converted to a compact format (TINF). These are embedded in firmware and loaded at runtime by a custom module that launches them as Zephyr threads. As a proof-of-concept, DAL has limitations: Cortex-M only, static stack allocation, no standard libraries, and no official support.

q3vm, is a lightweight C-based virtual machine from Quake III Arena. q3vm compiles C code to bytecode, which is embedded in firmware and runs in isolation. It lacks kernel API and standard library support, and its default stack size is too large for constrained devices, but it is modifiable and well-documented.

uPy is a minimal Python 3.4 runtime for microcontrollers, enabling easy scripting with Zephyr integration (GPIO, I2C, etc.). Scripts run in isolation but lack multi-interpreter support. Widely used in industry, it is a strong candidate for exploring scripting in *IoT*.
These solutions are then tested against each other in some computational tasks, as well as testing the update time and size of the update, and lastly, also checking the power consumption. However, the key takeaways can be seen already in their first performance test

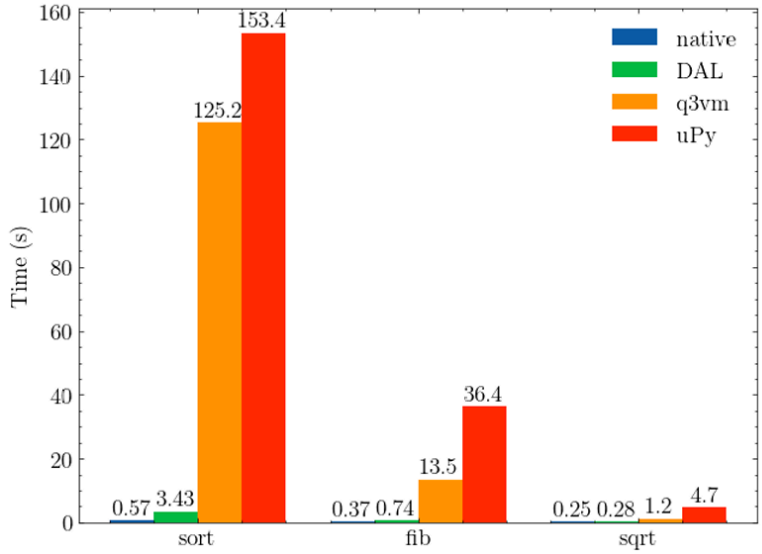

Figure 2.1: Performance based on bubblesort, Fibonacci seq., and calculating a square root *[16]*.

The paper makes a clear and concise conclusion of their findings: *"Dynamic code loading proved to be the best solution when performance is of significant importance, with a maximum penalty of 6x. All of the approaches showed massive reductions in the update time compared with the traditional update mechanisms, with reductions around 30x due to the reduced file sizes. Power consumption tests revealed no surprises where the solutions with higher overhead consumed more power*[16]*."*

Another research paper *[17]* with a focus on a small, compact OS called "TinyOS". Unlike the objective in the previous paper *[16]*, this paper focuses more on low-energy devices, while the other paper was more targeted towards microcontrollers in general. The update method used in the modified TinyOS is incremental and modular via the delta-based updates. The dynamic TinyOS solution outperformed the base solution by improving the performance in terms of updates, with the energy consumption and memory footprint, which resulted in a smaller run-time overhead. However, the approach also offers flexibility as a result of the ability to load modules dynamically.

The following research paper *[18]* looked at the most common methods currently being researched for reprogramming embedded devices when deployed.

	Mechanism						
	VM	**PIC**	**Reloc.**	**OS Protection**	**Kernel Modification**	**Kernel Replacement**	**Loose Coupling**
Maté [11]	•			•			•1
TOSBoot [12]						•	
SOS [13]		•			•		•
Contiki [1]			•			•	•
RETOS [14]			•	•	•		•
Darjeeling [15]	•			•			•
SenSpire [10]			•		•		•
Enix [16]		•2					

1 Only if user-specified instructions are omitted.

2 Kernel-supported PIC

Figure 2.2: The different OS's and mechanisms examined in the research paper *[18]*.

Depending on the use case, an argument could be made that each solution has a place, such as *VMs*, where the CPU overhead is not ideal for power-saving devices. Other approaches, like those used in Enix and SOS, have practical limitations due to hardware constraints or high runtime costs. A major issue across most systems is the lack of OS protection, which means a bad update can crash the whole device. Improving the reliability and safety of these update mechanisms will be key to making them more widely usable in real-world embedded systems.
Native code update methods that use relocatable code are the most efficient for long-term use because they avoid runtime overhead once the update is loaded. However, they can still introduce significant CPU costs when distributing updates across a network. Some systems, like SenSpire, avoid this by pre-relocating code, but that only helps with single-device updates.

The research paper *[19]* focuses on embedded devices with limited resources, with a focus on dynamic code loading. *PIC* is the method used so that the code will run regardless of its placement in memory, which is also referred to as dynamic linking. Non-PIC or static linking is where addresses have already been reserved. The paper aimed to make an enhanced *PIC* version, in which they succeeded with by both increasing the performance and minimizing the code size.

The solution required customization of the assembler and linker tools to support the changed "code" they refer to as the *"data-text relative concept"*. Usually, the compiler accesses the *Global Offset Table (GOT)*, which the authors replaced with "data-text-relative", which reduces the cost of memory by two lines for every time data is accessed. This results in only one instruction being used, thereby performing better in the tests. In the evaluation, they used Dhrystone, a benchmark designed to measure the CPU performance of embedded devices and as a way to quantify their performance gains from their *enchanced-PIC*.

2.3.1 Summary

Based on these different research papers, highlighting and optimizing performance or update methods for embedded devices will all be summarized in the following table *Table 2.2*.

Paper	Main Takeaway	Update Method	Metrics
Zephyr (2021) *[16]*	Dynamic app loading using a *SDK*, that enables efficient modular updates with low overhead.	Dynamic loading	**Overall:** time, size, **Memory footprint:** Flash, RAM, power consumption.
Dynamic TinyOS (2010) *[17]*	Modular, type-safe updates with low overhead.	Delta update	Latency, memory, power
Run-Time Reprogramming (2020) *[18]*	Relocatable code is best long-term; *VMs* suit rapid changes.	Native code	Energy consumption, OS protection, CPU costs
Bare-metal (2018) *[19]*	Enchanced-*PIC*	Static PIC + dynamic linking	Code size, Dhrystone benchmark, e.g., CPU performance

Table 2.2: Summary Table of the four research papers described in this section.

As seen in the *Table 2.2*, depending on the circumstances and goal of the project, the update method, and the evaluation metrics, a wide variety of procedures are employed to optimize a certain aspect. Although *[18]* does recommend only using VMs for short-term projects.

2.4 Fundamentals of Compilers

The C programming language is well known as one of the pillars of modern programming because of its efficiency and how much it has been implemented. This fact is also true in the world of embedded systems, which is why the report will be based around the C programming language. Another crucial aspect is the ability to compile the code instead of languages that are interpreted, such as Python.

2.4.1 Compiler

The compiler is, in essence, a translation process, although there is more to it as it can be divided into three main processes: *Preprocessing*, *Assembling*, and *Linking*, which at the end produces the executable output file as shown in the *Figure 2.3*. However, a few temporary files are also created in these processes to help facilitate the transitions.

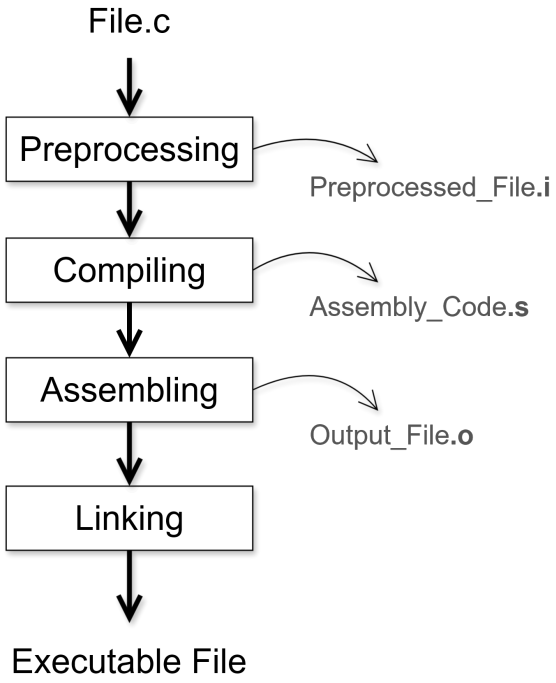

Figure 2.3: The compilation order in C.

In the preprocessing stage, three main actions are performed. The first is the removal of all the comments in the source file. Second is the inclusion of the code from the header files with the `.h` files, which contain the function declarations and the macro definitions. Thirdly, replace all macros with their values. All this combined creates the `.i-file`, which the compiler processes *[20]*.

In the compilation process, the code is translated to assembly code within the file type ".o". This assembly language acts as a low-level, human-readable instruction set for the processor, using operations like mov, push, or jmp. The resulting `.s` file represents this translation that forms the basis for the next step.

The assembler then takes over, converting, i.e., assembling the code into object code. The object code represents pure machine code, i.e., binary. The binary output stored inside the `.o` file, however, for external references such as calls to functions, has placeholders.

The final step is linking, where all the object files are consolidated and the references within the files are resolved. In static linking, all the required code and the libraries used are bundled directly into the final executable at compile time. This creates a self-contained file which easy to deploy because of its simplicity and portability. The main disadvantage with this approach is that if there were to be an update to any part, even a library, it would require a complete recompilation as mentioned earlier, as well *2.2.0.3: Update Execution Architecture*.

Dynamic linking utilizes a different approach in this step, which defers the inclusion of these eternal libraries until runtime. The program instead contains symbolic references to the shared libraries, which are loaded into memory based on need. This results in a reduced file

and allows for faster updates, but also introduces more runtime overhead and a dependency on having the correct versions of everything.

2.4.2 Memory Allocation

When looking further into how memory allocation works for variables and functions. Usually, it is divided into five categories: stack, heap, Uninitialized data (*Block Started by Symbol (BSS)*), initialized data, and text segment. A simple example of how location impacts the assembly code would be whether a variable has been defined globally or locally. The memory allocation is often shown, with one end representing the "high address" and the other "low address" as seen in the *Figure 2.4*. High and low addresses refer to how they are stored in memory, such that "`0xFFFFFFFF`" is a high address, while "`0x00000000`" is a low address.

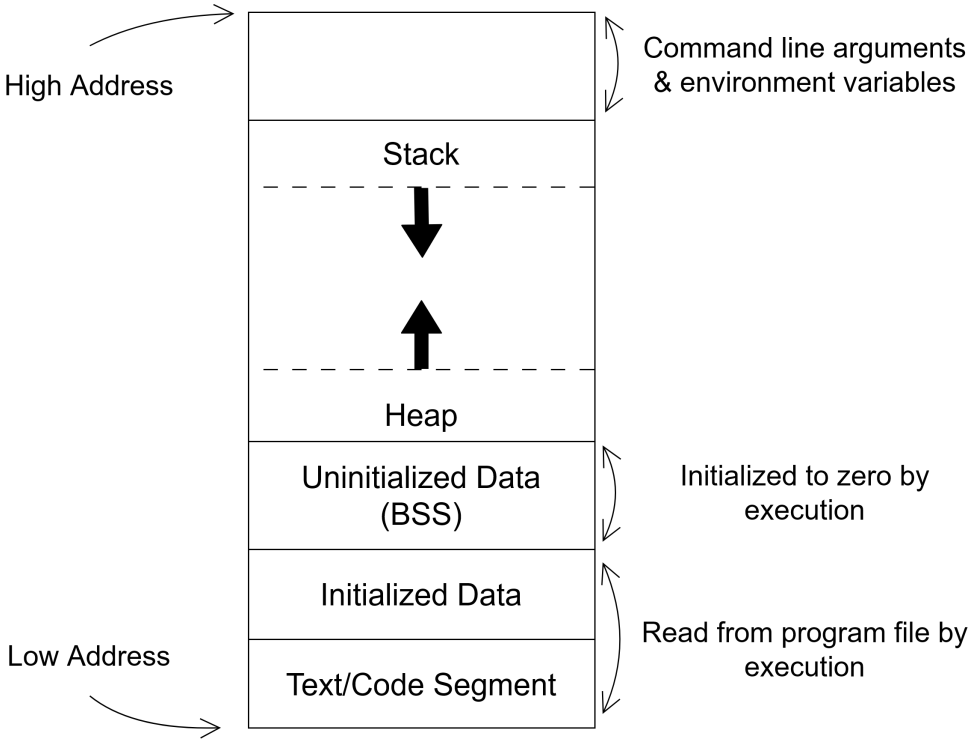

Figure 2.4: The hierarchy of memory allocation in C.

The stack stores variables with short life spans, e.g., local variables within functions. Stack allocation is where temporary data is stored, and it follows the LIFO principle. This principle allows for quick allocation and deallocation of memory as a feature. This, in turn, makes it efficient with function calls *[21]*.

Heap memory is often referred to as free memory. It stores variables that need to be accessed multiple times or have a longer life span. Unlike in the stack, where the compiler takes care of the memory allocation automatically, the heap allocates and deallocates memory using built-in functions such as `malloc(), calloc(), realloc(), and free()` during program execution. Heap memory is often used for data structures like trees, linked lists, and other types used for dynamically allocated variables. The heap, therefore, offers some flexibility in terms of memory block allocation *[21]*.

The uninitialized data segment contains all the local variables defined statically. However, as

seen in the *Figure 2.4*, the memory space allocated to them is initialized to zero at the start of the program. The initialized data segment refers to static, often global, variables that have been explicitly told to be initialized by the programmer and is stored in the binary. The code segment stores the executable code as it contains the actual instructions of the program *[22], [23]*.

2.4.3 Firmware File Types

Although some file types have been briefly mentioned in the previous *Fundamentals of Compilers* Section. However, this section explores the `.elf`, `.hex`, and `.bin` file types in more depth. Other file types exist, but these are some of the most commonly used in this field with embedded devices.

`.elf`

"Executable and Linkable Format", otherwise known as ELF, which is used for storing binaries, libraries, and core dumps in Linux and Unix-based systems. The ELF file is divided into two parts, a header and the ELF data. The first part of the header starts with four unique bytes: `0x7F 0x45 0x4c 0x46`, which translates to the letters E, L, and F as the file's identifier. Inside the header, the program headers are located that provide the information for how the segments of the executable file should be loaded into memory when the program executes.

The structure is similar to the one described earlier in *Memory Allocation Figure 2.4*, with some of the common sections being the `.text` for the instructions, `.data` for storing the initialized global and static variables, and the reserved space of the `.bss` for the uninitialized global and static variables *[24], [25]*.

`.hex`

`.hex` files are made up of hexadecimal data in ASCII text with addresses, data, and checksums. It is also referred to as Intel HEX, because of the checksums within the file, it automatically has a built-in error detection, and also contains the address information, however, the file size is often larger than a `.bin` as a result of metadata and the ASCII encoding *[25], [26]*.

`.bin`

The binary file (`.bin`) with a format of just raw binary numbers, and is more compact as there is no overhead in the form of metadata or checksums. It is therefore often the `.bin` that is sent when there is an *OTA* update *[25], [26]*.

2.4.4 Tools

This subsection explores some of the most common tools for *Delta updates*. The two most commonly used **Xdelta3**, and **BSDiff**, which utilizes two different approaches with their respective encoding algorithm. **Xdelta3** uses VCDIFF, which is a known data compression algorithm, while **BSDiff** uses a suffix sorting combined with a custom-made diff algorithm to detect similarities in the data. *[27], [28], [29]*.

2.5 Project Delimitation

This section summarizes the *2: Problem Analysis* chapter, such that a more concise and clear focus on the firmware update can be derived. Defining the objective, challenges, and assumptions for the firmware updates, and synthesizing these elements, leads to the creation of a 'Final Problem Statement'.

2.5.1 Summary

At the start of the analysis, the focus was on low-energy devices and how devices might need to conserve or make use of their time awake efficiently, such that only a small amount of the battery was consumed, even with updates. With this in mind, understanding the different aspects of updating methods was the next step. Based on this knowledge, researching the current *SoA* and commenting on the tradeoff of these methods and their focus became the next stepping stone. In the last part of the analysis, understanding how the compiler and memory allocation were described, as well as discussing some of the common file types for firmware updates.

Firmware updates are an essential part of the long-term "longevity" of these remote devices, which are in bandwidth-limited areas. Since most of the devices are running on battery power, efficient use of energy consumption is a crucial element, where the focus in this report lies in creating a firmware update that does not require too much processing power or time.

2.5.2 The Objectives

In this project, a direct correlation between the file size for the update with the processing and energy consumption is assumed. Based on this assumption, the objective is to reduce the firmware update file size. However, as discussed in the *SoA*, although using dynamic linking is preferred, it is not the easiest solution to make, which is why some opt for virtual machines. The second objective should focus on *Usability Factor* as an ease of use, such that it will be easier to make these firmware updates for embedded devices from the developer.

2.5.3 Memory Management

Delta Updates are a great tool for reducing the size of the update, which is why it is very commonly used by others. However, depending on how the changes to the firmware are applied, even a small change could cause changes to multiple lines, which causes the update file to be larger than necessary. It might not even be the fault of the firmware developer, as it can happen from changes to the compiler code itself, or from updates to the "base functions" of a device. With everything normally just being stacked on top of each other, even a small addition or removal of an array entry can offset most of the data in the program. This, of course, in turn means that *Delta Updates* just notices that all of those memory-entries have changed, ignoring that they might just be offset, and creates a large file with all the changes.

2.5.4 Final Problem Statement

Based on all these challenges and assumptions, it is now possible to define the Final Problem Statement with the following statement:

How can a system utilizing *Memory Management* to minimize data differences in firmware versions, increase the efficiency of *Delta Updates*, be created in a developer-friendly manner?

Solution Analysis 3

This chapter is used to set up the solution design, which includes the scenario expectations of what operations can be performed. This will therefore also include the limitations or constraints that might apply. At the end, these requirements will set for in a structured format such that it will be easy to evaluate the solution in the later chapters.

3.1 Scenario Classifications

This first section is dedicated to the different scenarios for what a firmware update might contain. The intention behind these scenarios is to incrementally increase the complexity of the scenarios while also being similar to what an update might contain in reality.

Five basic update scenarios with increasing complexity/size will be described, for further clarification on what exactly is needed for a possible solution to be able to do.

3.1.1 Scenario One - Update a Variable

The first task is to update a variable, as that should be the simplest and easiest change to make. The purpose of this change is to mimic the ability to create small changes in an embedded device. This change could be an interval change in how often it should send some data.

3.1.2 Scenario Two - Update an Array

In the second scenario, the focus is on updating an array, which also includes changing the size of the array. By adding another entry to an array, a "new" memory location needs to be added and taken into account. Depending on how it is implemented, it might impact larger parts of the code allocations in memory as a result. Mitigating big changes in memory allocations should be achievable, either by having a buffer zone or change the whole array's placement in memory to avoid larger changes.

3.1.3 Scenario Three - Update a Function

Updating a function in the code poses another spike in complexity, as a change in a function will likely change the size of the function and possibly also the addresses that the function occupies, as hinted at in the previous scenario. Minimizing relocations should be a priority, as this will impact the updated file size as it can create artificial changes in the sense that some code has been relocated.

That is why scenario three of updating a function should ideally also be able to change the whole function's placement in memory, as that will that cause the least amount of changes to the update file.

3.1.4 Scenario Four - Add a New Function

Similar to the previous scenario of changing a function, however, this scenario has the prerequisite of having sufficient space in terms of memory. Adding a new function scenario is based on the possibility that a device might need another feature, or maybe a law that would require some sort of logging or authentication.

3.1.5 Scenario Five - Change Function Calls

The fifth and final scenario looks at changing which functions are being called in the code. This could be switching from one function to another, e.g., replacing a basic function with an improved version or adding a call to a new logging feature. Even though the change might be small in the code, it can potentially affect other parts, like the memory addresses where the calls point to. These small address changes can cause extra updates in the binary, which might increase the size of the update file. This scenario helps show how even simple changes in code flow can have a bigger impact on how the update is applied to the device, as it essentially demonstrates the redirecting for the flow of execution.

3.2 Solution Considerations

This section delves into the main considerations in regards to memory control, the approaches with dynamic linking, following the *Usability Factor* aspect, and structured data. Based on these thoughts behind these concepts, a System Design can be procured.

3.2.1 Memory Management & Dynamic Linking

By having more or full control of where everything in the program is placed in memory, it is entirely in the hands of the developer how much is changed, and how. Full control over memory placements (named as *Memory Management* going forward) not only makes sure that you only need to transmit the actual difference, but keeping everything in the same place (as much as possible) should also increase firmware robustness to some degree. In either case, you are expected to know the location on the data currently on the device, but errors can always occur, so taking out the potential randomness from data which is not under your control, should improve the odds that whatever you are expecting in a specific address is actually where you expect it to be, even if the device has missed a couple of updates.

Padding & Stacking

- **Stacking**
  - Is memory efficient.
  - Big risk of having to move things around.
- **Padding:**
  - Creates headroom, when itself or its “downstairs neighbor” increases in size.
  - Requires more available memory.
  - Is still essentially *Stacking*, just with added buffer.

With both of these basically being *Stacking* they both create an inherent problem with everything shifting whenever something needs to move. This is of course based on the idea that they keep adhering to their *Stacking* "nature", instead of it just being from the very first version of the firmware.

Backup & Repurpose

A way to be able to update a device could potentially be to keep constant track of exactly where everything is on a binary level, and then creating a process which let you write to specific addresses on the device. This way you can add the new code "behind" the old code, and then just update the "Address File" to look at the new address. Using this approach you can also have the "Address File" link back to the old version, in case of failure of some sort, giving some rollback possibilities.
By keeping close track of memory-addresses you can make sure to add the code updates "behind" the old code, and then just update the *Dynamic Linking* table to look at the new address. Using this approach you can also have the *Dynamic Linking* table link back to the old version, in case of failure of some sort, giving some rollback possibilities. Of course this creates a never-ending tail of outdated code, which is a problem.

Then on the development side, to work around the never-ending tail, these addresses will be tracked closely, so that you can repurpose old space in later updates, after having confirmed that the newer version (which is stored elsewhere) is stable. At some point, an update to some other part of the device will be able to be written where the outdated code from something else was being stored.

As the chance of the new pieces of code taking up the exact same amount of space is close to non-existent, this approach needs to meticulously keep track of both the beginning and the end of the code, as the potential gaps in between used sections are supposed to be written to when possible.

It also creates a problem where old segments can only be overwritten by new segments of equal or smaller size, unless you have two unused segments besides each other, ready to be overwritten. Then, creating the unfortunate potential of running out of space, not because it is all being used, but because there is nowhere with a large enough "gap" for an update to write to. This state of memory clutter can still be overcome by sending a complete firmware in the old-fashioned manner, which is what this project is trying to avoid. At least having to potentially send a full update once in a while is better than every time, unless you are looking for a solution to this problem, because full updates is just plainly not possible for one reason or another 1.

Shift & Append

With everything stored in an *Dynamic Linking* table, another approach could be a local defragmentation of the device's storage. Tracking the positions of everything closely, when updating a functionality, it would be possible to just delete said functionality in memory, and then shift all memory after it, to where the now deleted functionality used to be. If you then enter the new code at the very end of the memory, the device would be back at a state where there was no gaps in the memory at weird places.

This approach needs you to update the address of every single part of the code that is being shifted, and not just the code that is being updated. This creates a higher risk of critical failure, although it arguably should not be too big of an issue, as the shift would be the same offset for each section. Increasing in complexity if updating more than one thing at a time.

A Hybrid Approach

By making a hybrid between the two approaches above, you can optimize the wasted space from *Backup & Repurpose*, and the high computation needs from *Shift & Append*. This comes at the cost of more complexity, though, as you then need to keep track of important details from either version.
It does potentially remove the need for keeping track of "broken pieces" from the *Backup & Repurpose* approach, as it would be possible to just leave the pieces of code not needed anymore, until space reaches whichever threshold is set for the device to run defragmentation.

Write updates to large enough available sections, if possible, or append if no large enough outdated section is available. Then with defragmentation being "optional", it creates a lot of flexibility in potential setups, for different needs. There could just be certain thresholds on when to defragment the device memory, like the first time an update is too large to fit in one of the unused segments, a time interval, or it could be based off of manual triggers, being sent out before an update. All of these come with their own drawbacks, which can be left completely up to the developer of a specific device, depending on their needs and hardware. But more options also comes with more complexity and with more things to go wrong.

3.2.2 Usability Factor

When the basis functionality of the system is in place, the project should attempt to make it easier to develop firmware using this solution, and can be thought of as a part of the project that focuses on the *Quality of Life (QoL)* of the solution. This is a wide topic which will be divided into multiple sub-categories, as there can be many different aspects to making the solution more intuitive, user-friendly, and better structured, including the potential use of scripts and databases.

All of these categories will go under the umbrella term *Usability Factor* going forward, to make it clearer that when wordings like "Ease of Use" and *QoL* are being used, that it is used in more general terms.

File Structure

Hiding files that the developer does not need to use makes it clearer which files they need to focus on, and makes it less overwhelming to get into. Thus, resulting in a better *Usability Factor*, because of the simplification that will make it more manageable to use.

Code Syntax & Boilerplate

Working towards having easily readable syntax and clearly named "interactables" goes a long way in making a system approachable by new developers, and the same applies to the amount of boilerplate code necessary to use said system.

Automation

Building automation for pieces or the entirety of the solution, so the developer can better focus on making an update, without being interrupted or confused by anything that does not directly concern them. This could all be made from small scripts and *Makefiles*, to bigger automated "building platforms" which could potentially even hold the history for the firmware, and take care of **everything** outside of the development of the actual functionality that the developer wants to implement.

Flexibility

Instead of being forced to be used in just one way, strive to make as much as possible flexible and reusable. Adding in options for several approaches to development, like freely choosing *Memory Allocation Padding* (also enabling *Stacking* by setting it to 0).

Examples of possible flexibility options:

- Memory Allocation Padding
- Number of Backups
- Defragment Memory Trigger Options
- Defragment Memory Interval
- Easy add-in of microcontroller/board, for auto-including their default functions and memory allocations
- Auto-compile several variants of the same code for several boards
- Auto-compile a series of programs, based on a list of scenarios and their changes

3.2.3 Structured Data

Storing *Structured Data* with different user-defined settings of the solution, storing the entire *Memory Allocation* structure of an implementation, or even the entirety of the implementation itself, is potentially a very powerful tool. This comes in many forms, with databases, *JSON*, *CSV*, and *XML* likely being the most widely used approaches to storing the data. With properly structured data, there is an almost limitless number of possibilities.

This can make it much easier to deal with many things, such as troubleshooting, building functionality for a *User Client GUI* of some sort, or visualizing data for analysis. In an extreme edge-case even enable to ability to pick and choose from **all** the earlier developments, available boards and option presets to make an "*Easy-Bake Franken-Ware*" that **just works** (maybe even in a *Drag'n'Drop GUI*.)

With this flexibility in mind, including structured data in the solution seems like a rhetorical question, especially considering the implications it can have on making the solution more user-friendly in several ways. This will likely happen in the form of a database, for extra flexibility in data arrangement and avoidance of potential read/write limitations from only one process being able to handle a "physical file" at a time.

3.3 Validation Specifications

Based on the focus of the project found in *2.5: Project Delimitation* and the previous section with the *3.2: Solution Considerations*, the validation specification for the solution should be defined in a manner that would make it easy to evaluate the success of the solution.

3.3.1 Update Optimization

Based on the overall scenarios described in *3.1: Scenario Classifications*, assessment of update optimizations should be based on how well the scenarios in the following tables have been fulfilled.

Memory Management

To get the system running, the main functionality that everything else is build upon is being able to put everything into a specific addresses in memory, so this is the obvious functionality to begin with.

With the *Memory Management* scenarios (identified by the prefix `M`), all but the very first scenario, will expect memory allocation is being controlled by system. All scenarios outside of the *Memory Management* group are working under the same assumption.

Scenario	Name	Description
M1	Hold Symbol	Put a variable or function in the program into a specific address.
M2	Move Variable	Move a same-size variable to another address.
M3	Move Array	Same as *M2* but a bigger reallocation.
M4	Move Function	Same as *M2* and *M3* but bigger yet again.
M5	Automatic Move	The system automatically detects conflicts, and corrects.

Table 3.1: Summary table of different scales of memory reallocation.

While on a surface level it can seem a bit redundant to include *M3* and *M4* after having done *M2*, but these are done also to look at the byte-differences of the compiled firmwares, both as a sanity check and for context.

Firmware Functionality Changes

These tests (identified by the prefix `F`) is the natural next step, which represents common changes a developer might make to a program in “normal development”.

Scenario	Name	Description
F1	Update Variable	Changing the value of a single variable.
F2	**Update Array**	**The F2 entries is about different array updates.**
- F2a	Move Entry	Moving an entry to a different position of the array.
- F2b	Add Entry	Adding an entry, and thereby increasing size of the array.
F3	**Update Function**	**The F3 entries are about different updates of existing functions.**
- F3a	Reduced Size	Reduced code size to represent optimization.
- F3b	Increased Size	Increased size of function. - Should trigger a compilation error
- F3c	F3b + Memory Move	Same as earlier scenario, but should now compile, as it has been moved to a address with sufficient space.
F4	**New Function**	**The F4 entries are about adding new functions to the system.**
- F4a	Add Function	Added a new function which is not used.
- F4b	Use New Function	Now the new function is used in the program.
F5	**Code Postioning**	**The F5 entries are the about positional changes in the code.**
- F5a	Variable Header	Changing the position in the development code *Header*.
- F5b	Variable Code	Changing the position in the main development code.
- F5c	Array Header	Changing the position in the development code *Header*.
- F5d	Array Code	Changing the position in the main development code.
- F5e	Function Header	Changing the position in the development code *Header*.
- F5f	Function Code	Changing the position in the main development code.
F6	**Code Comments**	**The F6 entries are about comments in code and positions.**
- F6a	Global Scope	Adding in a comment in the global scope.
- F6b	In Function	Adding in a comment in a function.
- F6c	In Array	Adding in a comment in an array between entries.

Table 3.2: Summary table of functionality change specifications.

Dynamic Linking

These tests (identified by the prefix `DL`) is the last of the natural steps, before going onwards to the the *Usability Factor* side of the project. These represent different functionalities tied with *Dynamic Linking*, with the *DL3* scenarios being for what will from this point be referred to as *Dynamic Sequencing*. Those last two scenarios is for making it more efficient to add or change the order of function-calls in the firmware, without having to make changes inside the `main()` function and thereby "pushing function bytes around".

Scenario	Name	Description
DL1	Add Function	Add a new function to the *Command Table*. - Requires *F4a*
DL2	Use Function	The new function is called via *Command Table*. - Requires *DL1*
DL3	**Dynamic Run**	**The DL3 entries are Dynamic Sequence changes.**
- DL3a	Change Sequence	Changes the sequence in which one of the commands is called.
- DL3b	Add Function	Add new function to sequence of calls. - Requires *DL1*

Table 3.3: Summary table of general *Dynamic Linking* specific updates.

Size of the Update

As discussed in *2: Problem Analysis*, the file size also impacts multiple aspects: the transmission of the update, the computational cost for the embedded device to process the update, and naturally, the file size itself. To evaluate the file size, *Xdelta3* and *bsdiff* use two different algorithms while also being commonly used *2.4.4: Tools*.
The evaluation will be against *Xdelta3*, *bsdiff* and a pure binary code file, comparing it against a version of the same firmware without the *Memory Management* aspect, which will be referred to as the "*Basic Linker*", with this project's implementation being called "*Custom Linker*".

3.3.2 Usability Factor

As discussed in the *2.5: Project Delimitation* and in the previous section *3.2: Solution Considerations*, the second part of project's focus is on the *Usability Factor*. *Usability Factor* will be set up similarly to the different scenarios.

Level of Success

Even with the *Usability Factor* being an integral part of the project as a whole, the upcoming sections will be approached on a subjective level of success. This was decided upon based on the fact that even if it's an integral part of the project, ease of use means **nothing** if the solution can not fulfil the basic functionality that it was meant for. After the basic functionality is present, the *Usability Factor* will be closely intertwined with most additional functionalities being implemented over time. These *Usability Factor* aspects can thus be set up as seen in the following tables: *Table 3.4 Table 3.5*.

File Structure

Level	Description	Validation
1	Basic functionality of the project, but usage is confusing and unapproachable.	Fail
2	File structure to make it viable to pick up as a new developer.	Partial Success
3	File structure to make it easy to pick up as a new developer.	Success
4	Partial extern tooling of the solution, so developer can have mainly their own files in their project, but might still be limited by some structure rules.	Great Success
5	Fully external tooling of the solution, so the developer can write their program almost just like they would write **any** program, not having to care about the quirks of development for embedded devices, and just "call" the tool when ready to compile.	Money Bags

Table 3.4: Table showing 5 levels of success of the code side of the *Usability Factor*

Code Syntax & Boilerplate

Level	Description	Validation
1	Basic functionality of the project, but it requires a lot of boilerplate code, and usage is confusing and unapproachable.	Fail
2	Naming conventions and workflows are annoying but approachable, with only some boilerplate code involved.	Partial Success
3	Level 1, but with automation for some of the more menial tasks - Or - Naming conventions and workflows are intuitive with little to no boilerplate code needed.	Success
4	Automation takes care of most of the setup and processing needed to work with this system, and naming conventions and workflows are intuitive with little to no boilerplate code needed.	Great Success
5	Full automation of everything not related to the development of the standalone functionality to be included in the firmware update, just from using this system.	Money Bags

Table 3.5: Table showing 5 levels of success of the code side of the *Usability Factor*

Proposed Solution 4

This is where the proposed solution will be described, but even with this being the "actual" design, there will still be some aspects that are described on a broader level. This is done as the project has the potential to be almost a lifetime project, if the basic *Memory Management* part just works, and having a large pool of achievements to reach is more educational than reaching a preset milestone.

4.1 Memory Structure Approach

As the solution is meant to be as streamlined and system-agnostic as possible, a sort of "build backwards" mentality was deemed a good approach. The idea behind it is that there can be large variances between the different *microcontrollers* and system implementations on how much "base data" is present, for whatever base-functionality they include. If memory is populated in a conventional, stack-like manner, any customized data would likely reside at different memory addresses across systems, even when using an otherwise identical program on two different *microcontrollers*.

Moving everything to the back, and "building backwards" means that the implementation can be at the exact same backwards offsets for every controller able to contain the program. All of this is done via custom-written *Linker Scripts*, which the compiler is then asked to use in the last step of compilation.

4.1.1 Quirks of Building Backwards

On a surface level, this does introduce a drawback for the `.bin` file type, as the file **always** will always be the full size of the available memory of the controller (*256KB* in our case). But as it's supposed to be on the device to begin with, and this is all done to be able to only transfer the **actual** difference, it becomes a non-issue, as all the empty space will not be transferred.

4.1.2 Padding vs. Stacking

This debate might not be needed for this design, as it is just a simple offset being added (or not) to a memory address, it is **very** easy to control and change via simple "project preference" setups at the most basic level of automation. Even with the main premise being purely to reduce update file size, giving developers the flexibility to adapt their solution to their needs, making it inconsequential.
And as much as the defragment approach is conceptually very interesting and potentially powerful, it seems more like the sort of thing that the *Bootloader* and/or update tool needs to account for, and not this system.

4.2 Dynamic Linking

For *Dynamic Linking*, the most important thing is to always know all addresses of the table, so with the "build backwards" approach in mind, the following solution seemed to make the most sense. The implementation is made so that the very last possible memory address able to contain an address *(4 bytes* of data), is meant to **always** be dedicated to holding the address of where the actual table containing the function links is placed. This is not meant for regular use, though, as it would be a waste of instructions to go to a pointer that shows another pointer, which shows you what you want. But it gives a surefire way of creating a fallback functionality, as it will **always** be known by the firmware, of where it can look for functions, if something should go wrong. These benefits also help with things like variables and constants.

So with the example of this project's implementation, from the *microcontroller* used here:

- *Flash* memory starts at address `0x08000000`
- *256 Kilobyte* of *Flash* memory available, meaning `0x40000` in *Hex-decimal*
- All addresses are *4 Bytes* in size (or *8 Hex-decimals*)

This means that the very last *Flash* memory address on the controller is `0x08040000`, and with the calculation `0x08040000 - 0x4` we know that the last possible address for the *Dynamic Link Table Pointer* is `0x0803FFFC`.

The address which is pointed towards will then hold an array with pointers to the actual functions, which can then be run.

Outside the robustness of this approach, it can also be beneficial for byte differences in updates, as a function call can always point towards the same *Dynamic Linking* table address, even if the actual function has been moved.

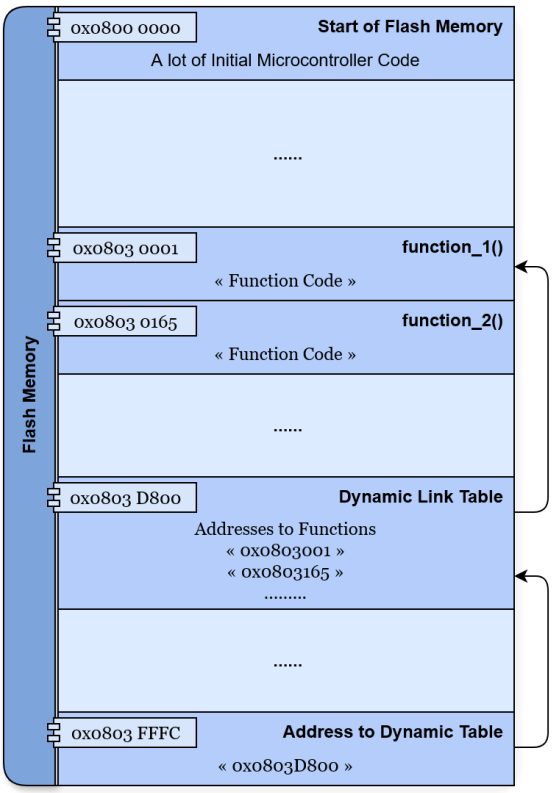

Figure 4.1: Diagram showing samples of memory addresses and "values" in the implementation.

4.2.1 Dynamic Sequencing

As explained in *3.3.1.3: Dynamic Linking*, adding *Dynamic Sequencing* functionality by creating arrays containing "function calls" is a potential way of editing what functions are called where, and the sequence in which it is done. Therefore, the system should include at least two such arrays: a one-time running sequence at device boot-up, and a sequence to run indefinitely inside a loop in `main()`.

4.3 Database

Based on the considerations in *3.2.3: Structured Data* and a project scale that does not expect a 10-year development cycle, a data structure will be designed with scalability in mind, divided into regions representing major development steps. It is unlikely that this project will need (and thereby implement) all of them, but as it is a fairly quick process to plan, it is here to illustrate the scalability and potential of this system design.

The different regions in the diagram are divided into color regions, each representing the addition of a major functionality of the system, but it is by no means necessary to implement in the represented order or combination. The implementation order of these will be almost completely dependent on a preferred focus/needs on further development of the system.

The base functionality of *Memory Management* is in the *Symbol*, *Symbol Version* tables (which could be combined if a simpler approach is wanted). Then what the group sees as most important, "next step functionality", is in the *Firmware* table, where general preferences for building the firmware are stored. The *Firmware* table is not connected to the mentioned *Symbol* tables, but it also does not need to be if all you want is "project preferences". For context, the *Symbol* naming is for all the possible defined variables, functions, and the like defined in the program itself, that someone would want to grab onto and "manually" place in memory at the position calculated by the system.

The color regions are as follows:

- **Blue Region**
  Easy Preferences, Backup/Fallback Functionality & Code Building Automation.
- **Purple Region**
  Version Control & Multiple Project Options.
- **Golden Region**
  Easy compilation of a single firmware for several different devices, and full dependency-fixing automation.

4.3.1 Structure Explanation

To keep it a bit short, this will mainly be explained at table level, as going through each individual datapoint is not gonna do much for understanding the overall concept. There is also a few tables which will be left out as their function should become kind of self-explanatory when the context of the other connected tables has been clarified.

To get an overall part out of the way from the start, **all** tables with "*Version*" in their name are solely there for version-history/backup functionality purposes. If those functionalities were to never be wanted, they are just an unnecessary increase in complexity for the system, and can just be merged with their "non-version" namesake. Knowing this, going forward, these tables will be explained "as one". This can be observed in the following *Figure 4.2*.

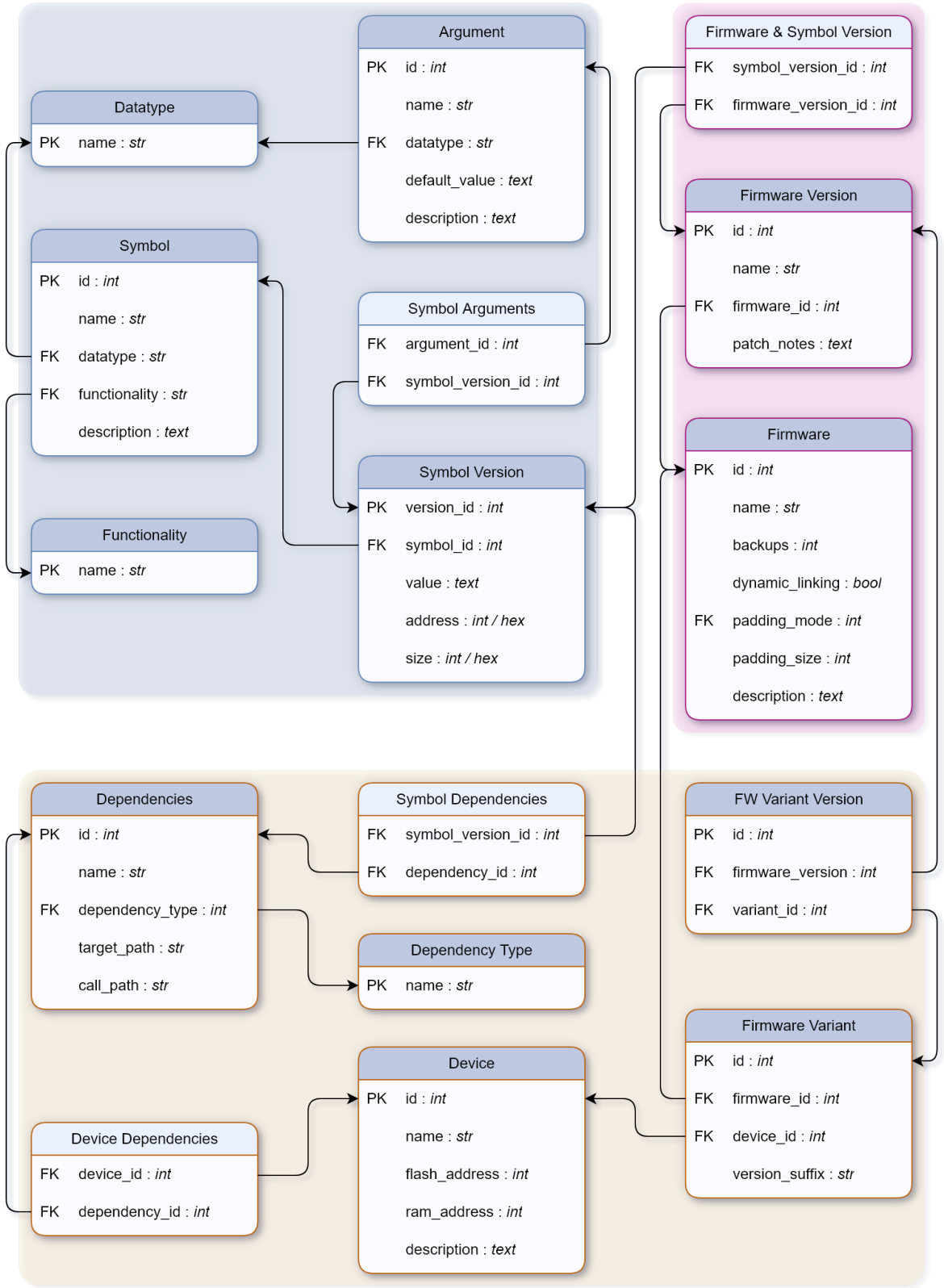

Figure 4.2: Diagram of database structure.

Secondly, there are a couple of tables in there just for good database practice of not holding too many repeated values of "the same thing". This includes the "*Functionality*" table, which is meant for telling "*Symbol Type*" (Variable, Constant, Function, etc.), and the "*Datatype*" table for *Return Types* (*Int, Char, Float*, etc.) of the a symbol. Lastly, it includes the "*Dependancy Type*" table, meant for identifying the nature of a dependancy, in the sense that it could be at compiler level, "code includes", *Linker Script* includes, standard library includes, or whatever else there might be.

Lastly, the tables with light nameplates (*Figure 4.2*). These are all "*Many to Many*" relational tables, which are there for cases where the same type of interaction can happen **many** times, or not at all, between the different tables/types, which for one reason or another need flexibility in this kind. An easy example is the "*Symbol Arguments*" in the blue region, which is set up that way to avoid creating a limitation on the number of parameters a function can have. So a function with 3 parameters would have 3 separate entries in "*Symbol Arguments*", one for each argument, and another *Symbol* containing a variable/constant would not have any entries in the "*Symbol Arguments*" table.

The more specific tables are:

- ***Symbol & Symbol Version***
  The mandatory first, and primary, table(s) of them all, when a database starts being implemented. This is where the memory address and size for each *Symbol* is being held, to be able to keep track of everything and potentially create automatic generation of the *Linker Scripts*. It also contains fields for containing the actual value (even function code) and other nice additions, if a high level of automation is wanted, where even the code-base could be build for you.
- ***Firmware & Firmware Version***
  This is likely the second most important table(s), as it is the one(s) holding the preferences for a project/firmware, for being able to easily automate aspects like the amount of fallback/backup versions, memory-padding, and the like.
- ***Device***
  For holding information on the different physical devices which is possible to develop on, for easier *Custom Linker* building for the same firmware between different devices.
- ***Firmware Variant / FW Variant Version***
  For creating "collaboration" between firmware and device options, so building the same firmware for several different device types can become as easy as "clicking a button".

This is scalable all the way up to *Easy-Bake Franken-Ware* without the danger of having to delete table data, as the next steps only interact with the former ones, without altering them. The only possible exception is from the argument that in a **very** early version, building the blue region only to keep track of memory addresses and preferences is overkill. If that approach is preferred, all but the *Symbol* and *Symbol Version* tables could be removed, and the two tables could be merged and trimmed in size.

The regions are also not "rigid structures", and separate functionalities from them can (within reason) be added in other steps. For example, if code-build automation is more important than the ability to have the same firmware easily compilable for different devices, the three *Dependency* tables needed for *Symbol Dependency* can be added even before the purple region is implemented.

4.3.2 Development Environment Scalability

The database implementation could be a local *SQLite3* database to have a separate database per project, for better project isolation, or just for smaller-scale development. Or a fully fledged server-based database for larger development organizations who want/need a more centralized approach that might even enable mixing and matching some partial software from one project to another.

4.4 Automation

With a proper data structure in place, and a bit of scripting to extract *Section* and *Symbol* data from the *Object Files*, the automation process should be fairly straightforward.

There are several available tools for analyzing compiled code, where it is just about finding the one (or several in conjunction) with it's output being consistent enough for a fairly simple script to extract all the necessary data. From extracting the needed data, the script should parse the data to the database to have maximum flexibility in what can be automated.

Then you can request the database for the rest of the necessary information, to start building the underlying structure to build the program you want compiled. With additional information being something like "project settings", for if you for example want *Dynamic Linking* to be enabled, or if *Memory Padding* is wanted, and how much padding there should be added. This also potentially includes controller/board specific data, like where the boards *Flash* memory is located, or even their full underlying compilation infrastructure, containing all their predefined base-functionalities. But this last part is very likely **very** time-consuming to set up, and is more written as a potential for the system if there were not a resource limitation set to a semester's worth of time.

When all the information needed is present, the script will start iterating through all the *Symbols* it found present in the compiled program, calculate the memory addresses based on padding offsets, statically size memory blocks, or whatever project preferences are present.
After all the preprocessing, the script will output a *Linker Script* to directly use for the last step of the compilation. There will likely be more auto-generated files involved, for things like supporting *Dynamic Linking*, but it largely depends on how far along the project will be by the end.

Validation / Results 5

This chapter will analyze the results of the implementation in this project and attempt to present them in a digestible manner for reflection.

5.1 Memory Management

As the main purpose of the project was to try and minimize differences in firmware updates by manually deciding where everything went in memory, this part is the obvious first step of validation that needs to happen.

Represented in the diagram on the right is only a few select examples from the implementation, which were picked as they give an overall representation of what is happening.

The notches on the top-left of each "memory entry", which is latching onto the *Flash Memory* bar, is representing the actual address of the corresponding *Symbol* in memory. The text in bold is an explanation of what is there, or an actual *Symbol* name (if the first letter is not capitalized). Lastly, whatever is written in the middle represents the data held by said *Symbol*.

As an example, first from the bottom is the very last 4 bytes of *Flash Memory* (with the starting address of `0x0803FFFC`), where the proposed design designates to keep the address of where the *Dynamic Link Table* starts. The value in the specified address is then exactly that, another address of where the *Dynamic Link Table* can be found (`0x0803D800`), which can then be found in the middle of the diagram, holding addresses to the actual functions.

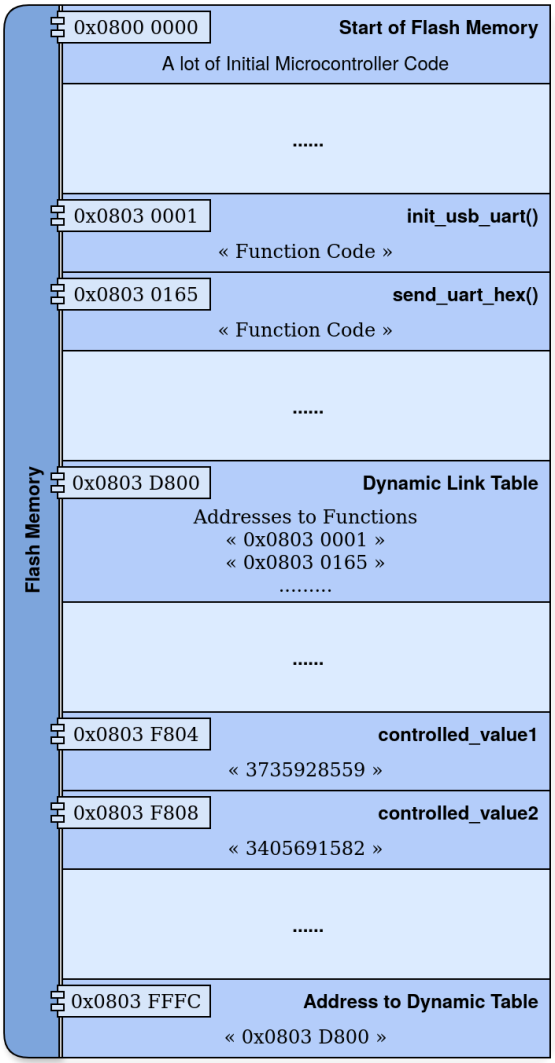

Figure 5.1: Diagram showing samples of memory addresses and "values" in the implementation.

Scenario *M5*, concerning automatic movement of a *Symbol* in case of overflow from an update increasing the size of it was never reached, do to time constraints.

5.2 Update Optimization

Memory Management via the project's *Custom Linker*, was just the vessel to drive the key part of this project, which was the optimization of the size of firmware updates. The performance for the *Custom Linker* will be compared to the *Basic Linker* variant, which is a *Linker Script* that outside of the *Memory Management* is identical to the *Custom Linker*. This was done to make sure that variances in *Linker Scripts* from different development platforms, etc. would not be able to skew the results, making them less meaningful. The baseline is a simple scenario that sends variables as messages via. *UART*.

These results were then examined using the tools of *bsdiff*, *Xdelta3*, and the pure binary difference (referred to as *Byte Diff* going forward) stored on the device. Three different firmware file-formats (`.bin`, `.elf`, and `.hex`) will be run through all three of the comparisons, for all the viable (and implemented) scenarios from *3.3: Validation Specifications*, making it a 9-way comparison between the *Custom Linker* and *Basic Linker* versions, across 23 scenarios. Generally, the `.bin` should be the smallest file as it is pure compiled code, while the other two contain various amounts of metadata as mentioned earlier in *2.4.3: Firmware File Types*.

The layout for the table of results (*Figure 5.2*) are as follows:

- **Column 1:** Has all the scenarios listed.
- **Column 2:** The file type used in the comparison on that row.
- **Column 3-5:** The differences in bytes for the *Custom Linker* variant.
  - Compared to its own *Custom Linker* Baseline
- **Column 6-8:** The differences in bytes for the *Basic Linker* variant.
  - Compared to its own *Basic Linker* Baseline
- **Column 9-11:** Percentile difference between the *Custom Linker* and *Basic Linker*

5.2.1 Overall

Most of the scenarios are improved with the *Custom Linker* approach compared to the *Basic Linker* approach, which can easily be identified with the color coding, where green indicates that the *Custom Linker* is better than the *Basic Linker*. The yellow color shows that the difference was the same for the updated file in both versions.

Deceptive Memory Management Scenarios

With the color-coded size differences, it seems mostly with *bsdiff* and the `move` scenarios, where the *Basic Linker* approach is more efficient, but this is deceptive, as the scenarios is specifically about "controlling memory locations", which is specifically not done with the *Basic Linker* versions. These scenarios were included to check functionality, and for gaining information on what a system like this would "cost" when having to move things around.

Scenario	Type	Byte Diff	bsdiff	xdelta3	BL + Byte Diff	BL + bsdiff	BL + xdelta3	%Δ Byte Diff	%Δ bsdiff	%Δ xdelta3
0_baseline	bin	0	135	60	0	140	63	0.0%	3.7%	5.0%
0_baseline	elf	0	141	60	0	144	66	0.0%	2.1%	10.0%
0_baseline	hex	0	141	60	0	140	66	0.0%	-0.7%	10.0%
0a_sanity_check	bin	0	135	65	0	140	68	0.0%	3.7%	4.6%
0a_sanity_check	elf	0	141	65	0	144	71	0.0%	2.1%	9.2%
0a_sanity_check	hex	0	141	65	0	140	71	0.0%	-0.7%	9.2%
DL1_add_command	bin	1291	446	580	1677	350	650	29.9%	-21.5%	12.1%
DL1_add_command	elf	127878	2660	10136	127654	2728	11224	-0.2%	2.6%	10.7%
DL1_add_command	hex	7288	978	1762	6710	1324	2134	-7.9%	35.4%	21.1%
DL2_use_command	bin	10398	716	961	11518	586	983	10.8%	-18.2%	2.3%
DL2_use_command	elf	137986	4257	16071	138701	4307	17630	0.5%	1.2%	9.7%
DL2_use_command	hex	24322	1859	5097	25444	2157	5778	4.6%	16.0%	13.4%
DL3a_change_sequence	bin	146	237	275	161	229	282	10.3%	-3.4%	2.5%
DL3a_change_sequence	elf	146	238	267	161	244	288	10.3%	2.5%	7.9%
DL3a_change_sequence	hex	270	362	451	300	358	474	11.1%	-1.1%	5.1%
DL3b_add_function_to_sequence	bin	1419	477	616	1818	420	682	28.1%	-11.9%	10.7%
DL3b_add_function_to_sequence	elf	127955	2691	10177	127715	2774	11247	-0.2%	3.1%	10.5%
DL3b_add_function_to_sequence	hex	7271	1230	1804	6784	1539	2157	-6.7%	25.1%	19.6%
F1_int_change	bin	1	144	71	1	150	73	0.0%	4.2%	2.8%
F1_int_change	elf	2	159	107	2	159	113	0.0%	0.0%	5.6%
F1_int_change	hex	2	152	76	2	156	83	0.0%	2.6%	9.2%
F2a_array_swap_item	bin	6	162	112	6	164	85	0.0%	1.2%	-24.1%
F2a_array_swap_item	elf	7	179	119	7	183	125	0.0%	2.2%	5.0%
F2a_array_swap_item	hex	10	170	117	10	173	124	0.0%	1.8%	6.0%
F2b_array_add_item	bin	2	149	79	2	157	79	0.0%	5.4%	0.0%
F2b_array_add_item	elf	3	168	113	3	170	119	0.0%	1.2%	5.3%
F2b_array_add_item	hex	5	159	81	5	160	90	0.0%	0.6%	11.1%
F3a_smaller_function	bin	81	222	172	1874	320	662	2213.6%	44.1%	284.9%
F3a_smaller_function	elf	127137	1951	9215	128184	2204	10248	0.8%	13.0%	11.2%
F3a_smaller_function	hex	2204	248	220	6443	888	2010	192.3%	258.1%	813.6%
F3c_bigger_function+move	bin	1629	469	741	1919	402	754	17.8%	-14.3%	1.8%
F3c_bigger_function+move	elf	125584	2413	10102	125278	2402	10519	-0.2%	-0.5%	4.1%
F3c_bigger_function+move	hex	5508	712	1769	7045	1021	2138	27.9%	43.4%	20.9%
F4a_add_function	bin	20	191	153	1558	304	626	7690.0%	59.2%	309.2%
F4a_add_function	elf	125201	2202	9397	125838	2562	11058	0.5%	16.3%	17.7%
F4a_add_function	hex	731	229	167	6679	758	1895	813.7%	231.0%	1034.7%
F4b_use_function	bin	10126	519	915	11275	530	947	11.3%	2.1%	3.5%
F4b_use_function	elf	135116	3842	15764	135453	4074	17423	0.2%	6.0%	10.5%
F4b_use_function	hex	25003	5483	8269	25802	5823	8783	3.2%	6.2%	6.2%
F5a_var_header	bin	0	135	64	0	140	67	0.0%	3.7%	4.7%
F5a_var_header	elf	99	182	152	83	180	158	-16.2%	-1.1%	3.9%
F5a_var_header	hex	0	141	64	0	140	70	0.0%	-0.7%	9.4%
F5b_var_code	bin	0	135	62	32	206	207	∞	52.6%	233.9%
F5b_var_code	elf	57	212	220	181	328	390	217.5%	54.7%	77.3%
F5b_var_code	hex	0	141	62	85	250	298	∞	77.3%	380.6%
F5d_array_code	bin	0	135	64	29	175	210	∞	29.6%	228.1%
F5d_array_code	elf	274	368	352	326	475	530	19.0%	29.1%	50.6%
F5d_array_code	hex	0	141	64	99	219	272	∞	55.3%	325.0%
F5e_func_header	bin	0	135	65	254	184	156	∞	36.3%	140.0%
F5e_func_header	elf	219	295	206	504	420	477	130.1%	42.4%	131.6%
F5e_func_header	hex	0	141	65	601	311	403	∞	120.6%	520.0%
F5f_func_code	bin	0	135	63	0	140	66	0.0%	3.7%	4.8%
F5f_func_code	elf	32467	296	379	32476	301	385	0.0%	1.7%	1.6%
F5f_func_code	hex	0	141	63	0	140	69	0.0%	-0.7%	9.5%
F6a_global_comment	bin	0	135	68	0	140	71	0.0%	3.7%	4.4%
F6a_global_comment	elf	0	141	68	0	144	74	0.0%	2.1%	8.8%
F6a_global_comment	hex	0	141	68	0	140	74	0.0%	-0.7%	8.8%
F6b_function_comment	bin	0	135	70	0	140	73	0.0%	3.7%	4.3%
F6b_function_comment	elf	0	141	70	0	144	76	0.0%	2.1%	8.6%
F6b_function_comment	hex	0	141	70	0	140	76	0.0%	-0.7%	8.6%
F6c_array_comment	bin	0	135	67	0	140	70	0.0%	3.7%	4.5%
F6c_array_comment	elf	1	153	76	1	157	82	0.0%	2.6%	7.9%
F6c_array_comment	hex	0	141	67	0	140	73	0.0%	-0.7%	9.0%
M2_move_variable	bin	10	168	149	0	140	69	-∞	-16.7%	-53.7%
M2_move_variable	elf	21	210	239	0	144	72	-∞	-31.4%	-69.9%
M2_move_variable	hex	7	168	190	0	140	72	-∞	-16.7%	-62.1%
M3_move_array	bin	13	189	108	0	140	66	-∞	-25.9%	-38.9%
M3_move_array	elf	21	197	216	0	144	69	-∞	-26.9%	-68.1%
M3_move_array	hex	5	163	146	0	140	69	-∞	-14.1%	-52.7%
M4_move_function	bin	182	213	224	0	140	69	-∞	-34.3%	-69.2%
M4_move_function	elf	263	302	340	0	144	72	-∞	-52.3%	-78.8%
M4_move_function	hex	1591	276	293	0	140	72	-∞	-49.3%	-75.4%

Figure 5.2: A comparison of the different scenarios, held against the baseline.

File-type Comparison

The simplest proof of earlier statement of `.hex` and `.elf` getting bloated by metadata, all the way down to 1 byte, is from the first of the functional update scenarios (`F1_int_change`) where the `.bin` has the lowest difference across the columns of methods used. In the third column of "*Byte Diff*"(pure `.bin`), the `int` change is equivalent to one byte, where the `.elf` and `.hex` seem to contain some metadata, which increases the size by one byte. This change might be small, but when compared to many of the other scenarios, this difference scales quite excessively.

An extra interesting point of interest, is that a comment inside of an array, actually creates a single byte of difference in the `.elf` file in scenario *F5d*. Becoming even more weird when nothing happens from comments in other scopes (other *F5* scenarios).

Deceptive Memory Management Scenarios

With the color-coded size differences, it seems mostly with *bsdiff* and the `move` scenarios, where the *Basic Linker* approach is more efficient, but this is deceptive, as the scenarios is specifically about "controlling memory locations", which is specifically not done with the *Basic Linker* versions. These scenarios were included to check functionality, and for gaining information on what a system like this would "cost" when having to move things around.

3 Approaches to Running Functions

Three different versions of running newly added functions were implemented, where it would be interesting to see what actual byte-difference they each cost to "activate". From time-constraints the calculations will be kept to only the most "pure" variants, using `.bin` and *Byte Diff*.

- F4b - F4a (Pure code)
  101026 - 20 = 10106
- DL2 -DL1 (Dynamic Linking)
  10398 - 1291 = 9107
- DL3b - DL1 (Dynamic Sequence)
  1419 - 1291 = 128

This shows that as hoped, the *Dynamic Linking* approach is very valid, especially if implemented with *Dynamic Sequencing* functionality.

5.3 Usability Factor

For the validation of the *Usability Factor*, it can be a bit harder to measure or show, as it is somewhat dependent on how "intuitive" the solution is to work with. The first part would be to make it somewhat manageable and transparent by showing the File Structure, followed by the Dynamic Linking and sequencing aspect of the code.

5.3.1 File Structure

The current file structure is as follows:

```
.
├── build
│   ├── objects
│   │   └── <all the object outputs>.o
│   ├── resources
│   │   └── <microcontroller specific files>
│   │
│   └── <compiled firmware>.bin /.elf /.hex
│
├── control
│   ├── available_commands.cpp    (Dynamic Sequencing)
│   ├── loop_commands.cpp         (Dynamic Sequencing)
│   └── startup_commands.cpp      (Dynamic Sequencing)
│
├── command.cpp
├── command.hpp
├── controlled.c                  (Definition of User Symbols)
├── controlled.h                  (Declaration of User Symbols)
├── controlled.ld                 (Custom Linker Script)
├── general_setup.hpp               (USER CODE)
├── general_setup.cpp               (USER CODE)
├── linker.ld
├── main.cpp                        (USER CODE)
└── makefile
```

The above structure is not taking project development-specific folders and files into account, as these would not be present in scenario where a developer would be using the system. The files without a description in brackets besides them, should not need to be interacted with by the user/developer, and should arguably have been "hid away". This is not the cleanest structure for newcomers to approach, but not the worst either, and with a bit of guidance it should be somewhat useable for users outside of this project.

The biggest plus to the current structure is that the 3 files for easy editing of available commands for *Dynamic Linking*, and the two implemented *Dynamic Sequencing* arrays, are to be found in their own separate folder.

5.3.2 Dynamic Linking

In the following *Figure 5.3*, all the different commands available for use in the program via *Dynamic Linking* are shown in the implementation.

```
////////////////////////////////////////////////////////////////////////////////////////
//  -  A list of commands to add to the overall available list of commands
//////////////////////
//  First "string" is just a calling-name,
//  Followed by the actual function (no brackets), type-cast to a void pointer
//
//  Build like the normal "inside of an array" in C++
//////////

{ "blink", (void*)toggle_led_pb5 },
{ "send_string", (void*)uart_send_string },
{ "send_hex", (void*)uart_send_hex },
{ "send_int", (void*)uart_send_int },
{ "send_array", (void*)uart_send_array },
// { "greet", (void*)greet },           // Scenario DL1
```

Figure 5.3: The different shared libraries.

5.3.3 Dynamic Sequencing

Startup Sequence

On boot of a device, there is usually commands and initiations that needs to be run, which can be done via this specific list of commands, which will be executed in the order that they are in the array in this file. These will execute once, before the device enters the infinite loop in `main()`. This list of commands can be seen below in *Figure 5.4*.

```
////////////////////////////////////////////////////////////////////////////////////////
//  -  A list of initial commands to run before the eternal main loop
//////////////////////
//  Is being executed in the order that they're entered here.
//
//  Build like the normal "inside of an array" in C++
//////////

{ "send_string", (void*)"Init done!" },     //  Baseline
{ "blink", nullptr },
{"send_string", (void*)"\n\n  -- Initiating Main Functionality --\n\n"},
{"fail_on_purpose", nullptr },
// {"greet", (void*)"Fiskepinde" },         //  Scenario F4c (add dynamic function) &  S2 (add function to
sequence)
{"send_string", (void*)"  -- After Fail thing --\n\n"},
// { "send_string", (void*)"Init done!" },     //  Scenario S1  (Change Sequence)
```

Figure 5.4: The initial commands run before entering the main loop.

Loop Sequence

After entering the eternal loop in `main()`, the list (from *Figure 5.5*) below, will be executed in order, over and over again.

```
////////////////////////////////////////////////////////////////////////////////////////
//  -  A list of commands to run in the eternal while-loop in the main function
//////////////////////
//  Is being executed in the order that they're entered here.
//
//  Build like the normal "inside of an array" in C++
//////////

{"send_string", (void*)"\n  -- Main Loop Cycle Started --\n"},

{ "blink", nullptr },

{"send_string", (void*)"Controlled Value 1 (via send_string) is:"},
{"send_string", hex_buffer1},
{"send_string", (void*)"Controlled Value 1 (via send_hex) is:"},
{"send_hex", (void*)controlled_value1},


{"send_string", (void*)"Controlled Value 2 (via send_string) is:"},
{"send_string", hex_buffer2},
{"send_string", (void*)"Controlled Value 2 (via send_hex) is:"},
{"send_hex", (void*)controlled_value2},

{"send_string", (void*)"Controlled Value 3 (via send_string) is:"},
{"send_string", int_buffer},
// {"send_string", "Controlled Value 3 (via send_int) is:"},
// {"send_int", controlled_value3},


// {"send_string", "Controlled Array 1 (via send_string) is:"},
// {"send_string", int_buffer},
{"send_string", (void*)"Controlled Array 1 (via send_array) is:"},
{"send_array", (void*)controlled_array1},
{"send_string", (void*)" -- Array DONE  --"},

{"send_string", (void*)"\n  -- Main Loop Cycle Ended --\n"},
```

Figure 5.5: This figure represents the main loop cycle for the program.

Discussion & Conclusion 6

6.1 Discussion

The discussion will mainly highlight some of the difficulties with the implementation and with some of the limitations, both in terms of time constraints, but also partly due to the linker script. Besides this, discussing the results would also be included here as well as the "Future Work".

6.1.1 Implementation

The first part of the discussion is focused on the implementation.

Database Implementation

Due to unforeseen complications in other parts of the project, the solution never reached a state where the database came into play, and therefore never got implemented. The current version of the system is at a point where the next step would be to make it more user-friendly with some form of automation. The files that a user would need to interact with are now somewhat easy to separate from the rest, making it easier to create scripts for automation from structured data.

Memory Management Options

For the current state of the project implementation, "padding or no-padding" and similar ideas have not been taken into account, as there was no automation and never really any risk of the microcontroller running out of memory. It seemed like a waste of time to have to meticulously calculate and pick addresses when it was just supposed to work on a basic level before it supposedly became automated to some degree. As such, it was somewhat out of the scope and therefore disregarded.

Linker Script Limitations

The current implementation of *Dynamic Linking* is an overly complicated variant that seems to create more complications than it fixes. It is limited to 0 or 1 argument functions, based on wanting to avoid boilerplate code and convoluted usage/syntax of the functionality. A significant amount of time has been spent on *Linker Script* approaches that did not feel like menial labor, which seems to not actually be possible with the current available compiler tool set (from *GNU Compiler Collection* at least).
With one of the very next steps of development likely being automation of that particular part, it feels like a waste of resources, and in hindsight implementation of a database to start development of the automation would have arguably been a better use of time.

Considering that all global variables and functions are visible as *Symbols* when analyzing a compiled *Object File*, it seemed like the obvious choice to just use the *Linker Script* to grab hold of those, and tell the linker to put them in a specific position in memory. This proved to be more difficult than anticipated, if not impossible.

The implementation is done via *Sections*, which means to be regions holding several *Symbols* that belong together as a sort of group, for one reason or the other. So to have **complete** control of where **everything** is being held in memory, each and every variable and function get a *Section* created for them in the *Linker Script*. This does not cost anything space-wise in the final compiled binary, as all this info is not present, although the `.elf` or `.hex` file type variants do hold information of those *Sections*. This is, as mentioned, because these two file types hold more metadata.

Worse than the *Linker Script* being a bit messy to work with, there is a much larger complication involved with this approach. It requires that you, in the code, when initializing everything, explicitly tell it that this new thing is supposed to be put into this *Section*.

This is done by adding `__attribute__((section(".section_to_put_it_into")))` behind the name, which adds **a ton** of extra keystrokes for **every single initialization** of variables, functions, and the like, but is also not very intuitive. Imagine having to convince someone that it's smarter for them to write:

```
int foo = 2;
```

They should be writing:

```
int foo __attribute__((section(".foo"))) = 2;
```

So this part instantly made it much more complicated to make a system where developers can "just write code" like they are used to, and single-handedly made us have to consider file-structure and general *Usability Factor* in a different light than earlier.

6.1.2 Results

This part is dedicated to some of the unusual results that were found, and what could cause their respective outcomes.

Limitations of bsdiff & Xdelta3

By using these tools, it seems that there is an inherent minimum file size and some overhead due to the metadata needed. This could clearly be determined by the `baseline` and `0a_sanity_check` scenario, although most notable with *Xdelta3*, with all the file types being the same size. However, it seems that this is also true for *bsdiff*, whose minimum size seemed to be at least twice the size of *Xdelta3* for the "small" changes.

Additionally, these versions were significantly larger than the pure *Byte Diff* version, and it could seem like a natural next step for a solution like this project, would be to look into creating a custom-made patcher. The overhead **is** of course from the fact that they need to know **where** to put the data, but it could seem like there is much more data on top of that.

With a system like this, there should be enough control of everything in memory, to be able to reduce the metadata overhead of patch tools like *Xdelta3* and *bsdiff*.

Comments in Code

During the testing, and how the code commenting affected the output files, which most would assume would not result in anything different, as it has usually been taught that comments are deleted from the final output file. However, that statement does not always hold true, as it was found that if the comment were located in an array, this would impact the difference in the delta update file.

6.1.3 Future Work

The next steps for the project should be concentrated on improving the *Usability Factor* aspects, as the design of it has already been laid out in the *3: Solution Analysis*, which makes it a natural progression choice. With a project like this, there are quite a lot of directions for what the focus should be long-term. It could lean towards a very flexible system that allows for various options for more types of memory management.

Another subject which this report has not focused on is the security considerations for using *Delta Updates*. This is another quite large direction in which the focus could be on adding support for a cryptographic verification of patches and memory integrity checks, or focus on secure delta update protocols to ensure tamper-resistance during transmission and application.

6.2 Conclusion

The primary goal of the project was to send a more "efficient" firmware update, which meant focusing on minimizing the file size by having greater control over the memory, i.e., *Memory Management*, with a secondary focus on *Usability Factor*, also often referred to as a *QoL* aspect or developer-friendly.

Based on the analysis of how compilers *2.4: Fundamentals of Compilers*, file types, different methods, and approaches, such as the current *SoA*, helped shape the direction of using *Dynamic Linking* with *Delta Updates*. With the scope narrowed to using these methods to design and a *Custom Linker* that will help conserve the battery on low-energy devices located remotely in bandwidth-limited areas.

By subdividing this goal into a part focused on the size of the update, and another on the *Usability Factor* aspect based on the research, which also highlighted that as a major drawback for why developers do not use a more efficient approach. To assess the problem, we assumed that there was a direct correlation between the file size and the battery consumption needed to process it.

To combat these restrictions, *Memory Management* combined with *Dynamic Linking* would be needed to design a functioning *Custom Linker* that could create smaller file sizes against some typical scenarios, which could be set up as a form of validation, which also included an evaluation format for the *Usability Factor*.

The *Custom Linker* was implemented successfully, such that all the scenarios could be achieved; however, concerning the *Usability Factor* requirements, it was lacking. Yet, with the *Memory Management* in the *Custom Linker* proved to decrease the file size in most cases, with the exception of the `moving` of code, and often when using *bsdiff*. However, if the focus is on `.bin` files, then the *Custom Linker* could be considered a success in achieving a smaller firmware file, thus lowering the battery usage required to apply the update.

Despite the shortcomings of the *Usability Factor* objective, a fully functioning *Custom Linker* has been implemented and does function according to the outcome of the results by improving the lifetime operation of the low-energy devices as a result of minimizing the file size.

List of Acronyms

BSS. Block Started by Symbol 11

Basic Linker. Project specific, identical comparison firmware, without @mm. 22, 31, 33, 49

Byte Diff. The amount of bytes inside a file that differs between two files. 31, 33, 38

Custom Linker. The project's solution. 22, 28, 31, 40, 49

Delta Update. Only transferring the data that differs, instead of a full program, during updates. 39, 40

Dynamic Linking. Program and external libraries are separated, and referenced at runtime. 5, 17, 22, 25, 33, 34, 35, 40, 52

Dynamic Sequencing. Having an array holding "function calls" to easily change function-order of firmware. 22, 25, 33, 34

EM. Energy Mangement 1

GOT. Global Offset Table 8

IoT. Internet of Things 1, 2, 6

Linker Script. Used for configuration of the final step of compiling code, which gives "locations" to everything 24, 28, 29, 31, 37, 38, 49

Memory Management. Controlling the specific addresses that things are put into memory 16, 20, 22, 31, 40

OTA. Over the Air 3, 5, 12

P2P. Peer-to-Peer 3, 5

PIC. Positional Independt Coding 3, 4, 6, 8, 9

QoL. Quality of Life 18, 40

RTOS. Real-Time Operating System 6

SDK. Software Development Kit 6, 9

SoA. State of the Art 6, 13, 40

Symbol. Overall term for definitions in a program (variables, functions, etc.). 26, 28, 29, 30

Usability Factor. Wider term used to refer to the solutions' overall ease of use. 13, 16, 18, 22, 23, 33, 38, 39, 40

VM. Virtual Machine 6, 8, 9

Wi-Fi 1, 5

Xdelta3. Commonly used tool for creating patch-files for @dl:pl. 22, 31, 38, 39

bsdiff. Commonly used tool for creating patch-files for @dl:pl. 22, 31, 33, 38, 39, 40

References

[1] S. Sinha, "Number of connected IoT devices growing 13% to 18.8 billion." [Online]. Available: https://iot-analytics.com/number-connected-iot-devices/

[2] "Key IoT Applications: Caburn Telecom (By CSL Group)." [Online]. Available: https://caburntelecom.com/where-iot-mostly-used/

[3] "How to optimize low-power operation for sensor applications." [Online]. Available: https://pt.farnell.com/how-to-optimize-low-power-operation-for-sensor-applications-trc-ht

[4] V. Struk, "Why IoT Firmware Update Matters: Expert Tips and Strategies." [Online]. Available: https://relevant.software/blog/iot-firmware-update/

[5] R. U. I/O, "Firmware Development and Update Strategies for Embedded Systems | by RocketMe Up I/O | Medium." [Online]. Available: https://medium.com/@RocketMeUpIO/firmware-development-and-update-strategies-for-embedded-systems-677e247b8c07

[6] T. Pasternak, "A/B updates. Over-the air software updates are an… | by Tal Pasternak | Medium." [Online]. Available: https://medium.com/@pasternaktal/a-b-updates-254e891d3d0b

[7] Google, "Use a canary deployment strategy | Cloud Deploy | Google Cloud." [Online]. Available: https://cloud.google.com/deploy/docs/deployment-strategies/canary

[8] GeeksforGeeks, "Static and Dynamic Linking in Operating Systems | GeeksforGeeks." [Online]. Available: https://www.geeksforgeeks.org/static-and-dynamic-linking-in-operating-systems/

[9] A. Upadhyay, "Static and Dynamic Linking Explained - Earthly Blog." [Online]. Available: https://earthly.dev/blog/static-and-dynamic-linking/

[10] Arm, "Arm Compiler for Embedded User Guide." [Online]. Available: https://developer.arm.com/documentation/100748/0621/Mapping-Code-and-Data-to-the-Target/Support-for-Position-Independent-code?utm_source=chatgpt.com

[11] C. to Wikimedia projects, "Position-independent code - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Position-independent_code

[12] Arm, "User's Guides for Keil C51 Development Tools." [Online]. Available: https://developer.arm.com/documentation/101655/0961/BL51-User-s-Guide/BL51-Introduction/Purpose-of-the-Linker/Combining-Segments?utm_source=chatgpt.com

[13] C. S. Erica Mixon, "What is OTA update (over-the-air update)? | Definition from TechTarget." [Online]. Available: https://www.techtarget.com/searchmobilecomputing/definition/OTA-update-over-the-air-update

[14] T. Azilen, "An Ultimate Guide to Peer-to-Peer Topology for IoT Networks." [Online]. Available: https://www.azilen.com/learning/peer-to-peer-topology-for-iot/

[15] Y. Busnel, M. Bertier, E. Fleury, and A.-M. Kermarrec, "GCP: Gossip-based Code Propagation for Large-scale Mobile Wireless Sensor Networks." [Online]. Available: https://arxiv.org/abs/0707.3717

[16] J. Oliveira and F. Sousa, "Reprogramming of Embedded Devices using Zephyr: Review and Benchmarking," in *2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*, 2021, pp. 484–489. doi: 10.1109/EuCNC/6GSummit51104.2021.9482443.

[17] W. Munawar, M. H. Alizai, O. Landsiedel, and K. Wehrle, "Dynamic TinyOS: Modular and Transparent Incremental Code-Updates for Sensor Networks," in *2010 IEEE International Conference on Communications*, 2010, pp. 1–6. doi: 10.1109/ICC.2010.5501964.

[18] R. Oliver, A. Wilde, and E. Zaluska, "Reprogramming Embedded Systems at Run-Time," *International Journal on Smart Sensing and Intelligent Systems*, vol. 7, pp. 1–6, 2020, doi: 10.21307/ijssis-2019-078.

[19] A. Sadek, M. Elmahdy, and T. Eldeeb, "Dynamic Code Loading to a Bare-metal Embedded Target," in *Proceedings of the 7th International Conference on Software and Information Engineering*, New York, NY, USA: ACM, 2018, pp. 1–6.

[20] "C Compilation Process." [Online]. Available: https://www.tutorialspoint.com/cprogramming/c_compilation_process.htm

[21] Leslie, "Memory Usage in C Programming: A Comprehensive Guide," *Medium*, May 2023, [Online]. Available: https://medium.com/@lsltry404/memory-usage-in-c-programming-a-comprehensive-guide-b20038647992

[22] A. Poddar, "Memory layout of C program - Naukri Code 360." [Online]. Available: https://www.naukri.com/code360/library/memory-layout-of-c-program

[23] Contributors to Wikimedia projects, ".bss - Wikipedia." [Online]. Available: https://en.wikipedia.org/w/index.php?title=.bss&oldid=1237337541

[24] Ajmewal, "Basics of ELF (Executable and Linkable Format) file | by Ajmewal | Medium." [Online]. Available: https://medium.com/@ajmewal/basics-of-elf-executable-and-linkable-format-file-88a516877356

[25] Syspcb, "Difference between BIN, HEX, AXF, ELF files - SYS Technology Co., Ltd.." [Online]. Available: https://www.syspcb.com/pcb-blog/knowledge/difference-between-bin-hex-axf-elf-files.html

[26] C. Evans, "The difference between HEX and BIN files in microcontrollers - Blog - Ampheo." [Online]. Available: https://www.ampheo.com/blog/the-difference-between-hex-and-bin-files-in-microcontrollers?srsltid=AfmBOoo-gn5ZN0nA46jvg5HGMHw-aB96qv2atRBeqEZi5nY9yEnEo8Sw

[27] jmacd, "GitHub - jmacd/xdelta: open-source binary diff, delta/differential compression tools, VCDIFF/RFC 3284 delta compression." [Online]. Available: https://github.com/jmacd/xdelta

[28] mendsley, "GitHub - mendsley/bsdiff: bsdiff and bspatch are libraries for building and applying patches to binary files.." [Online]. Available: https://github.com/mendsley/bsdiff?tab=readme-ov-file

[29] C. to Wikimedia projects, "VCDIFF - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/VCDIFF

Appendix A

A.1 Memory Addresses

```
There are 49 section headers, starting at offset 0x2ada8:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .isr_vector       PROGBITS        08000000 001000 000138 00   A  0   0  1
  [ 2] .text             PROGBITS        08000138 001138 002914 00  AX  0   0  4
  [ 3] .rodata           PROGBITS        08002a4c 003a4c 000300 00   A  0   0  4
  [ 4] .ARM.extab        PROGBITS        08002d4c 003d4c 000048 00   A  0   0  4
  [ 5] .ARM              ARM_EXIDX       08002d94 003d94 0000e0 00  AL 34   0  4
  [ 6] .preinit_array    PREINIT_ARRAY   08002e74 007000 000000 04  WA  0   0  1
  [ 7] .init_array       INIT_ARRAY      08002e74 003e74 000008 04  WA  0   0  4
  [ 8] .fini_array       FINI_ARRAY      08002e7c 007000 000000 04  WA  0   0  1
  [ 9] .data             PROGBITS        20000000 004000 0005a0 00  WA  0   0  8
  [10] .hex_buffer1      PROGBITS        200005a0 0045a0 00000b 00  WA  0   0  4
  [11] .hex_buffer2      PROGBITS        200005ac 0045ac 00000b 00  WA  0   0  4
  [12] .int_buffer       PROGBITS        200005b8 0045b8 000008 00  WA  0   0  4
  [13] .bss              NOBITS          200005c0 0045c0 00019c 00  WA  0   0  4
  [14] ._user_heap_stack NOBITS          2000075c 00475c 000604 00  WA  0   0  1
  [15] .ARM.attributes   ARM_ATTRIBUTES  00000000 007000 00002c 00      0   0  1
  [16] .link_table_[...] PROGBITS        0803fffc 006ffc 000004 00   A  0   0  1
  [17] .command_table    PROGBITS        0803d800 005800 000028 00   A  0   0  4
  [18] .startup_sequence PROGBITS        0803d828 005828 000030 00   A  0   0  4
  [19] .loop_sequence    PROGBITS        0803d858 005858 000088 00  WA  0   0  4
  [20] .controlled_[...] PROGBITS        0803f804 006804 000004 00   A  0   0  4
  [21] .controlled_[...] PROGBITS        0803f900 006900 000010 00   A  0   0  4
  [22] .controlled_[...] PROGBITS        0803f808 006808 000004 00   A  0   0  4
  [23] .controlled_[...] PROGBITS        0803f80c 00680c 000004 00   A  0   0  4
  [24] .main_loop_i[...] PROGBITS        0803f810 006810 000004 00   A  0   0  4
  [25] .init_usb_uart    PROGBITS        08010000 005000 0000b4 00  AX  0   0  4
  [26] .uart_send_char   PROGBITS        080100b4 0050b4 000038 00  AX  0   0  4
  [27] .uart_send_array  PROGBITS        080100f0 0050f0 00003a 00  AX  0   0  2
  [28] .uart_send_string PROGBITS        08010138 005138 00002c 00  AX  0   0  2
  [29] .uart_send_hex    PROGBITS        08010164 005164 00005e 00  AX  0   0  2
  [30] .uart_send_int    PROGBITS        080101c8 0051c8 000028 00  AX  0   0  2
  [31] .uart_send_n[...] PROGBITS        080101f8 0051f8 000030 00  AX  0   0  4
  [32] .init_led_pb5     PROGBITS        08010228 005228 000038 00  AX  0   0  4
  [33] .toggle_led_pb5   PROGBITS        08010260 005260 00001c 00  AX  0   0  4
  [34] .int_to_hexstring PROGBITS        0801027c 00527c 000076 00  AX  0   0  2
  [35] .int_to_string    PROGBITS        080102f4 0052f4 0000e4 00  AX  0   0  4
  [36] .comment          PROGBITS        00000000 00702c 00001e 01  MS  0   0  1
  [37] .debug_info       PROGBITS        00000000 00704a 00e820 00      0   0  1
  [38] .debug_abbrev     PROGBITS        00000000 01586a 003ccb 00      0   0  1
  [39] .debug_aranges    PROGBITS        00000000 019538 000650 00      0   0  8
  [40] .debug_line       PROGBITS        00000000 019b88 005dd8 00      0   0  1
  [41] .debug_str        PROGBITS        00000000 01f960 002577 01  MS  0   0  1
  [42] .debug_rnglists   PROGBITS        00000000 021ed7 0005d2 00      0   0  1
  [43] .debug_frame      PROGBITS        00000000 0224ac 000e34 00      0   0  4
  [44] .debug_loclists   PROGBITS        00000000 0232e0 0039ee 00      0   0  1
  [45] .debug_line_str   PROGBITS        00000000 026cce 000136 01  MS  0   0  1
  [46] .symtab           SYMTAB          00000000 026e04 0028b0 10     47 394  4
  [47] .strtab           STRTAB          00000000 0296b4 001457 00      0   0  1
  [48] .shstrtab         STRTAB          00000000 02ab0b 00029c 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), y (purecode), p (processor specific)
```

Figure A.1: A printout of the different sections and their addresses of the baseline firmware.

Figure A.2: A printout of `controlled_value1` being held on address `0x0803BEEF` in memory.

A.2 Firmware Functionality Changes

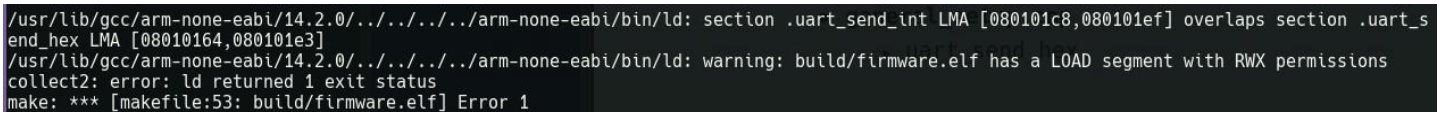

Figure A.3: A printout of compilation error from not having enough space.

Validation Run Setup

B

This is a walkthrough on how the different scenarios specifically differ in the validation process, and how to "get them there".

B.1 0 - Baseline

Well… it's the baseline… what is there to say?
Make sure that everything is "back in order", before changing to a new scenario, unless the formers steps are needed for the next step.

File Specifics

- **`main.cpp`**
  - Before **main loop**, make sure the call of command `greet` is commented out
- **`controlled.c`**
  - `controlled_value3`
    - Value of `0x8420`
  - `controlled_array`
    - Commented out at the top
    - Baseline contains 3 values
    - `998123` is last entry
  - `int main_loop_interval`
    - Commented out at the top
      Commented in at the bottom
  - `int main_loop_interval`
    - Commented out at the top
      Commented in at the bottom
- **`general_setup.cpp`**
  - `greet`
    - Commented out (is for **F4a**)
  - `init_usb_uart()`
    - Commented out at the top
      Commented in around the middle
  - `uart_send_hex`
    - Looks like

```
// general_setup.cpp
void uart_send_hex(uint32_t value) {

    uart_send_char('0');
    uart_send_char('x');

    for (int i = 7; i >= 0; i--) {
        uint8_t nibble = (value >> (i * 4)) & 0xF;
        uart_send_char(nibble < 10 ? '0' + nibble : 'A' + nibble -
10);
    }

    uart_send_new_line();
}
```

- **general_setup.hpp**
  - `void greet(void* msg)`
    - Commented out at the bottom (is for **F4a**)
  - `static int not_used`
    - Commented out at the top
      Commented in at the bottom
  - `void init_usb_uart()`
    - Commented out at the top
      Commented in at the bottom
- **base_linker.ld**
  - Commented out:
    - `.controlled_value1 0x0803BEEF : {KEEP(*(.controlled_value1)) }`
- **controlled.ld**
  - Addresses Fixed as:
    - `.controlled_value1 0x0803F804 : {KEEP(*(.controlled_value1))}`
    - `.controlled_array1 0x0803F900 : {KEEP(*(.controlled_array1))}`
    - `.uart_send_hex 0x08010164 : {KEEP(*(.uart_send_hex))} > COMMAND_TABLE`
  - Commented out:
    - `/* .greet 0x080103D8 : {KEEP(*(.greet))} > COMMAND_TABLE */`
- **startup_commands.cpp**
  - `send_string` with "Init done!"
    - Present, and is **first** command to run
  - `greet`
    - Commented out (is for **F4c** & **S2**)
- **available_coamands.cpp**
  - `greet`
    - Commented out (is for **F4c**)

Potentially compile a second one as a sanity check.

B.2 Memory Address Control

Remember for these, in the comparison table, that the `BL` version isn't using the **Linker Script**, so when they're shown as being far more efficient, it's because they're not actually moving anything around.

B.2.1 M1 - Hold Symbol

As all other scenarios are done using a lot of held addresses in the *Custom Linker*, this is done in the *Linker Script* for the *Basic Linker*.

- **`base_linker.ld`**
  - Addresses Fixed as (bottom of file):
    - `.controlled_value1 0x0803BEEF : {KEEP(*(.controlled_value1)) }`

This is holding on (and moving) an `int` with the value of `0x8420` so it's supposedly 4 bytes that is changed. Although, with the original memory position being empty now, the last `0` is likely **not** changed, and it should be 3 bytes. Times two of course, as it's moved **from** somewhere **to** somewhere else.

B.2.2 M2 - Move Variable

- **`controlled.ld`**
  - Addresses Fixed as:
    - `.controlled_value1 0x0803F800 : {KEEP(*(.controlled_value1)) }`

Same *Byte* "behaviour" as explained for *M1* should be seen here.

B.2.3 M3 - Move Array

- **`controlled.ld`**
  - Addresses Fixed as:
    - `.controlled_array1 0x0803F908 : {KEEP(*(.controlled_array1)) }`

B.2.4 M4 - Move Function

- **`controlled.ld`** - This part is technically the **M3** scenario
  - Addresses Updated as:
    - `.uart_send_hex 0x080103f0 : {KEEP(*(.uart_send_hex)) } > COMMAND_TABLE`

B.2.5 M5 - Automatic Move

Automation is not implemented, so isn't present here.

B.3 Functional Updates

B.3.1 F1 - Single Value Update

- **`controlled.c`**
  - `controlled_value3` (Swap value `0x8420` with `0x8720`)

B.3.2 F2 - Array Updates

F2a - Swap Item

- **`controlled.c`**
  - `controlled_array` (3 values, `998123` in the middle)

F2b - Add Item

- **`controlled.c`**
  - `controlled_array` (4 values, `998123` is last, before new entry, new entry is `5324`)

B.3.3 F3 - Function Updates

F3a - Smaller Function

This version is smaller than the original.

- **`general_setup.cpp`**
  - `uart_send_hex`

Change `uart_send_hex` into:

```
// general_setup.cpp
void uart_send_hex(uint32_t value) {

    char hex_buffer[10];
    int_to_hexstring(value, hex_buffer);

    uart_send_string(hex_buffer);
}
```

F3b - Larger Function

This version is LARGER than the original.
And therefore needs to be moved in memory, which is isn't being, meaning that compilation should fail.

- **`general_setup.cpp`**

- `uart_send_hex`

Change `uart_send_hex` into:

```
// general_setup.cpp
void uart_send_hex(uint32_t value) {

    char hex_buffer[10];
    int_to_hexstring(value, hex_buffer);

    uart_send_string(hex_buffer);
    uart_send_string("In Hex thingy");

    uart_send_char('0');
    uart_send_char('x');

    for (int i = 7; i >= 0; i--) {
        uint8_t nibble = (value >> (i * 4)) & 0xF;
        uart_send_char(nibble < 10 ? '0' + nibble : 'A' + nibble - 10);
    }

    uart_send_new_line();
}
```

F3c (and M3) - Larger Function

Requires *F3b*

The larger version from *F3b* needs to be moved in memory, and should then be able to compile again.

- **`controlled.ld`** - This part is technically the **M3** scenario
  - Addresses Updated as:
    - `.uart_send_hex  0x080103f0 : {KEEP(*(.uart_send_hex)) }  > COMMAND_TABLE`

B.3.4 F4 - Add New Function

F4a - Add Function Link

- **`general_setup.hpp`**
  - Commented in:
    - `void greet(void* msg)`
- **`general_setup.cpp`**
  - Commented in the entire function of:
    - `void greet(void* msg)`
- **`controlled.ld`**
  - Commented in:
    - `.greet          0x080103D8 : {KEEP(*(.greet))}          > COMMAND_TABLE`

F4b - Use New Function

This is for "in code" use, and not using *Dynamic Linking*.

- **`main.cpp`**
  - Before **main loop**, make sure the call of function `greet` is commented in, so it's used by the program.
    - `greet("Fiskepinde");  //  Scenario F4b`

    OBS! There's a *Dynamic Linking* version close by, pick the right one.

B.3.5 F5 - Code Positioning

Move *Symbols* around in code to be earlier/later positions in a header or "real code" file, to see compiled byte behavior.

F5a - Variable Header

Defining variables in headers is bad practise, so none of the implementation's header-files contain variable definitions which are used, so this variable is there, just for testing purposes (actually being used despite the name).

- **`general_setup.hpp`**
  - Almost top of the file, comment in:
    - `static int not_used = 666;  // F5a`
  - Around middle of the file, outcomment:
    - `static int not_used = 666;  // Baseline`

F5b - Variable Code

- **`controlled.c`**
  - Around the top of the file, comment in:
    - ```
      const
      int main_loop_interval __attribute__((section(".main_loop_interval")))
      = 1500000;  // F5b Scenario
      ```
  - At the bottom of the file, outcomment:
    - ```
      const
      int main_loop_interval __attribute__((section(".main_loop_interval")))
      = 1500000;  // Baseline
      ```

F5c - Array Header

Did not have time to go through the same pain as making a use for a "header variable" in *F5a*, so this did not happen…

F5d - Array Code

- **`controlled.c`**
  - At the top of the file, comment in the **full array**:
    - ```
      const                                                    int
      controlled_array1[4] __attribute__((section(".controlled_array1")))
      ```
  - Around top of the file, outcomment in the **full array**:
      ```

  - `const int controlled_array1[4] __attribute__((section(".controlled_array1")))`

F5e - Function Header

- **`general_setup.hpp`**
  - Almost top of the file, comment in:
    - `void init_usb_uart() __attribute__((section(".init_usb_uart")));  // F5e`
  - Around middle of the file, outcomment:
    - `void init_usb_uart() __attribute__((section(".init_usb_uart")));  // Baseline`

F5f - Function Code

- **`general_setup.cpp`**
  - Almost top of the file, comment in **the entire function** of:
    - `void init_usb_uart()  // F5f`
  - Around middle of the file, outcomment **the entire function** of:
    - `void init_usb_uart()  // Baseline`

B.3.6 F6 - Code Comments

As you can't just outcomment comments, these scenarios will have to be made with actually cutting out the complete line in code.

F6a - Global Scope

- **`main.cpp`**
  - `// Test comment`
    - Somewhere outside of functions

F6b - Function Scope

- **`main.cpp`**
  - `// Test comment`
    - Somewhere inside of `main()`

F6c - Function Scope

- **`controlled.c`**
  - `controlled_array` (3 values, added `// 5324` at third entry)

B.4 Dynamic Linking

***DL1 - Add Function to* Command Table**

Requires *F4a*

- **`general_setup.hpp`**
  - Commented in:

  - `void greet(void* msg)`

- **`general_setup.cpp`**
  - Commented in the entire function of:
    - `void greet(void* msg)`

- **`controlled.ld`**
  - Commented in:
    - `.greet 0x080103D8 : {KEEP(*(.greet))} > COMMAND_TABLE`

- **`available_commands.cpp`**
  - `greet` is commented in as the last on the list.

***DL2 - Use Function from* Command Table**

Requires *DL1*

This is for "in code" use, and not the **Sequence Controlled** lists.

- **`main.cpp`**
  - Before **main loop**, make sure the call of function `greet` is commented in, so it's used by the program.
    - `call_command("greet", (void*)"Fiskepinde"); // Scenario DL2`

    OBS! There's a non-dynamic version close by, pick the right one.

B.4.1 DL3a - Change Sequence of Commands

- **`startup_commands.cpp`**
  - `send_string` with "Init done!" present, and is **last** command list
    - Remember to comment out the identical one further up

B.4.2 DL3b - Add New Function to Sequence

Requires *DL1*

- **`startup_commands.cpp`**
  - `greet` is commented in as the third last of the list.