

Fabian Schlager

# **Implementation of a reliable update protocol for SmartOS**

**Master's Thesis**

zur Erlangung des akademischen Grades  
Master of Science

eingereicht an der  
**Graz University of Technology**

Betreuer

Univ.-Prof. Dipl.-Inf. Univ. Dr.rer.nat. Marcel Carsten Baunach  
Eng. Leandro Batista Ribeiro

Mitbetreuer

Institute for Technical Informatics

Graz, March 2020



---

## Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

---

Date

---

Signature



# Abstract

The goal of this thesis was to enable SmartOS to receive, load and execute applications and system components at run-time. This was achieved by implementing a protocol to communicate over UART with a backend server that compiles and links modules on demand. The loose-coupling architecture implemented in this thesis uses trampoline functions to decouple previously direct function calls. The added overhead due to indirections in function calls was measured on an MSP430 and amounts to about four times as many CPU cycles per call compared to direct function calls.



# Contents

<b>Abstract</b>	<b>v</b>
<b>1. Introduction</b>	<b>1</b>
<b>2. Problem Statement and Motivation</b>	<b>3</b>
2.1. Making a Monolithic Kernel Modular . . . . .	3
2.1.1. Memory Management . . . . .	4
2.2. Changing Code at Run-time . . . . .	5
2.3. Update Communication Protocol . . . . .	5
2.4. Contributions in this Thesis . . . . .	7
<b>3. Related Work</b>	<b>9</b>
3.1. Operating Systems for Embedded Devices . . . . .	9
3.1.1. Contiki . . . . .	9
3.1.2. Contiki-NG . . . . .	10
3.1.3. FreeRTOS . . . . .	10
3.1.4. RIOT OS . . . . .	10
3.1.5. Zephyr Project . . . . .	10
<b>4. Design and Concept</b>	<b>13</b>
4.1. Device . . . . .	13
4.1.1. Device Performance Classes . . . . .	13
4.2. Module Server . . . . .	15
4.3. Update Protocol . . . . .	16
4.3.1. Communication . . . . .	16
4.3.2. Synchronization . . . . .	17
4.4. Managing Memory . . . . .	17
4.4.1. Flash Memory . . . . .	18

4.5. Managing Modules . . . . .	19
4.5.1. Module & Load Information State . . . . .	20
4.6. Loose Coupling . . . . .	21
<b>5. Implementation and Realization</b>	<b>29</b>
5.1. Module Server . . . . .	29
5.1.1. Modules . . . . .	31
5.2. Memory Management . . . . .	32
5.3. Module Management . . . . .	35
5.4. Update Protocol . . . . .	36
5.4.1. Preparation & Setup . . . . .	38
5.4.2. Module Request . . . . .	42
5.4.3. Dependency Installation . . . . .	44
5.4.4. Relocation & Transmission . . . . .	44
5.4.5. Module Installation & Loading . . . . .	46
5.5. Storing in Flash Memory . . . . .	46
<b>6. Evaluation and Analysis</b>	<b>51</b>
6.1. Size Analysis of a Module . . . . .	51
6.2. Performance Analysis . . . . .	51
6.2.1. Analysis of TTBL based calling . . . . .	52
6.2.2. Clock cycle analysis . . . . .	55
6.3. Time Analysis . . . . .	58
6.3.1. Modular Installation . . . . .	59
<b>7. Summary</b>	<b>61</b>
<b>8. Conclusion and Future Work</b>	<b>63</b>
8.1. Outlook . . . . .	63
8.1.1. Module Server . . . . .	63
8.1.2. Preserving State . . . . .	64
8.1.3. Module IDs & Versioning . . . . .	65
8.1.4. Update Protocol . . . . .	66
8.1.5. Memory Manager . . . . .	66
<b>A. Glossary</b>	<b>71</b>
<b>B. Code</b>	<b>77</b>



<b>Bibliography</b>	<b>83</b>
---------------------	-----------



# 1. Introduction

Modularity is key when it comes to the rapid development process of modern software engineering. Applications and system components need to be updated in a reliable way, without disrupting normal operation. This thesis discusses the modification of a real-time kernel for embedded devices to explore ways of dynamically loading components at runtime.

The kernel this work is based on is called MCSmartOS<sup>1</sup> and is being developed at the Institute of Technical Informatics at Graz University of Technology. It “aims on novel design concepts for compositional applications in future embedded systems” [17].

Reloading system components, for instance drivers, during run-time introduces new problems. The new driver version might not be binary compatible to the old one as symbol addresses might have changed. Running applications which use a driver that was updated might access invalid memory when trying to call a driver’s function, as they are linked to an older version. It is impracticable or, in certain contexts even impossible to link applications to new driver versions on demand, due to limited resources and overhead introduced. To solve this, the operating system’s components should be loosely coupled. Loose coupling is described as “[reducing the] number of assumptions two parties make about each other when they exchange information” [14]. In the case of system level programming, the assumptions made by two parties could be compile-time availability of information, for instance physical location in memory.

As loosely coupled components, applications are no longer hard linked to their drivers, but rather call functions using redirections provided by the operating system. That way, the operating system can automatically redirect

---

<sup>1</sup>Multi Core Sustainable Modular Adept Real-Time Operating System

## 1. Introduction

---

applications to the new driver's function addresses, without the need to relink or even restart them.

This work presents the detailed concept and implementation of an update mechanism described in the paper "Towards Automatic SW Integration in Dependable Embedded Systems" [3].

To summarize, these are the goals this thesis attempts to achieve:

- Compile modules for SmartOS on a host machine.
- Establish a protocol to transmit modules from a host machine to a device.
- Transmit the compiled module in as few bytes as possible.
- Load modules on a device during run-time.
- Enable module function calls across modules in a dynamic environment.

In the first part, the problems and challenges mentioned above will be analyzed in more detail. Afterwards, existing solutions to these problems are presented and discussed. Further loose coupling mechanisms can be found in other embedded operating systems, for instance Contiki, which will be described as well. Following that, a description of the way these problems can be solved in MCSmartOS is given.

## 2. Problem Statement and Motivation

To achieve the goal of supporting component updates during run-time, three problems need to be addressed:

- The current implementation of the SmartOS kernel is monolithic: Making it modular involves module and memory management, as well as modifying task load routines to support dynamic components. Section 2.1 goes into details about this.
- Individual parts of the operating system are not loosely coupled: Allow sharing and reuse of drivers and other system components between applications, as described in Section 2.2.
- Updating components during run-time requires a protocol between device and the other party serving new components. The requirements of this protocol are described in Section 2.3.

### 2.1. Making a Monolithic Kernel Modular

The SmartOS kernel and many other embedded kernels are designed in a monolithic way. Every component, application, driver, library and the kernel itself are compiled and linked into one binary image that is then flashed onto its target chip. This simplifies many things, as there is no need for dynamic address resolution or memory management, since both addresses and memory used are known at compile time. The downside of this design is the low degree of flexibility. Components cannot be loaded or exchanged at run time. In case any of the system's parts needs to be updated, the whole binary image has to be rebuilt and programmed onto the hardware,

## 2. Problem Statement and Motivation

---

even parts that weren't directly affected by the changes. This increases the difficulty and effort required to maintain the system.

It is especially problematic for real-time systems, as flashing the new image halts and then resets the CPU and therefore all running applications. On top of that, all state will be lost in the process, if not somehow preserved before the reset. The time it takes to program a new image depends on the hardware, its interfaces and the size of the image. Thinking of critical infrastructure, for instance control systems in power plants, downtimes to apply updates might not be acceptable due to risk of injury, damage or decrease in quality of service or financial loss due to stopped production lines.

Developers of applications also need access to either the whole source code or at least relocatable object files of all parts of the operating system to be able to build complete images every time they deploy their work. This not only increases turn around times, it also makes it harder to independently develop components in parallel. It could also be of interest to not release the full source code of an operating system, but rather provide developers with only the necessary parts to extend it.

The ability to update and reload only specific parts of the system at run time compared to building a complete image, flashing and rebooting the system to load the changes decreases the time developers have to wait before being able to test their work, allowing for faster on-device test iterations. On top of that, changing one component will not affect other components running on the system, as long as no interfaces shared between them change. Thus the state of unaffected parts of the system remains intact.

### 2.1.1. Memory Management

For monolithic builds, one of the linker's responsibilities is to verify that the binary image and all of its sections fit into their respective memory regions of the hardware. Naturally, it can do so only after being configured for the target hardware. When dynamically loading components at run time, the linker cannot be aware of memory constraints and requirements. Different devices can run the same system in different configurations, for

instance using different drivers or applications. Therefore memory needs to be managed dynamically. The system needs to be able to allocate and free blocks of memory and cannot simply rely on the linker to verify that there is enough free space at compile time. Efficient memory management is not trivial, as problems such as fragmentation need to be handled.

At its current state, the SmartOS kernel does not offer dynamic memory management. All objects are allocated in a static way, either on the stack or by reserving memory regions via a linker script and directly addressing them in code. Newer versions of modules might have changed in size, then the system needs to be able to keep track of free and reserved memory regions, to determine if enough memory is available for updates and to place new code and data safely without overwriting existing regions.

## 2.2. Changing Code at Run-time

As many modern real-time operating system do, SmartOS allows for multiple tasks to run concurrently in a cooperative multitasking environment. These tasks might want to rely on each other, for instance by communicating distributing data or other results. They might make use of the same data structures or algorithms, at which point it becomes beneficial for storage capacity to share code between tasks, as only one version of the code needs to be stored on device. All of this introduces dependencies which make it non-trivial to exchange parts of the code on the device without breaking references to the old parts, as the current implementation links all dependencies tightly together.

How this tight coupling between tasks and other components can be loosened is described in Section 4.6.

## 2.3. Update Communication Protocol

For distributing updates, two parties will communicate with each other according to a well-defined protocol. In the case of this thesis, one party will

## 2. Problem Statement and Motivation

---

be the device initiating communication and requesting an update, called *device* in the following. The other side provides modules and responds to these requests and will be called *server*.

The main requirement for the update process is, that neither the device nor the server should ever be left in a state from which they can't recover. From this, the following potential issues can be identified:

- Unexpected events (for instance crashes) during installation: If either the device or the server experience a crash during an update process it can be left in an undefined state. Both should be able to restart into a working condition, so that the update process can be performed again without additional actions.
- Data can get corrupted during transfer, for instance through bit flips. The protocol needs to be able to detect that, so that a new transfer attempt can be started. Under no circumstances should the device try to start a module that could be corrupted, as this can lead to an unrecoverable state.
- Incompatibilities between installed modules: Since the design allows to install not only applications but also drivers and libraries, there could be a situation where applications require different versions of the same driver. Therefore, drivers can't simply be updated on the device without further compatibility checks, as this could lead to crashes in applications expecting a different version. An *interoperability check* will verify that versions are compatible. See Section 4.1.1.
- Incompatibilities between modules and kernel: The system also needs to be able to determine if a module is compatible with the kernel version running on the device before installing it. This is also part of *interoperability*.
- Smart memory management: Installation and removal of modules can lead to fragmented memory. The memory manager should be able to detect and fix this at a convenient time, for instance without interrupting running real-time tasks. The memory manager also needs to be able to report if not enough memory is available for an installation. This is part of the so-called *pluggability check*. See Section 4.1.1.
- Safely removing modules: When removing a module, it has to be made sure beforehand that no other modules depend on it. Otherwise other



applications could be compromised, resulting in crashes. On top of that, the implementation should check for orphaned dependencies and remove them as well, so it frees up as much memory as possible.

- Reliable update protocol: The protocol between device and server should be reliable and stable. It should not be susceptible to timing issues.

An issue that is not treated in this work is instabilities due to bad code in the modules. It is the developers responsibility to make sure their module does not break or cause corruptions.

Also not in the focus of this thesis is the migration of state data between module updates. In an ideal implementation, modules would automatically migrate their state to its new memory layout before they start running again. For more information on this matter refer to [Section 3](#).

## 2.4. Contributions in this Thesis

This thesis describes and implements a possible way of achieving run-time updates on embedded devices. It does so by implementing basic necessities, such as a driver for flash memory, a memory manager for ROM and RAM, modularization in the kernel, including loose coupling. Our proof of concept shows that it is possible to perform modular updates and resolve dependencies.

Due to the limited scope of the master thesis, some aspects are left as future work, such as fault-tolerance techniques and more sophisticated memory management. Some complex problems of modularization, like shared global variables, (function) pointers into modules or state migration are not covered in this work.



## 3. Related Work

### 3.1. Operating Systems for Embedded Devices

There are several papers and projects with similar goals to SmartOS. This section describes a selection and analyzes if and how they try to solve the topic of providing modular updates.

#### 3.1.1. Contiki

Contiki [6] is an open source operating system for embedded devices. Its source code is available on GitHub<sup>1</sup> under a 3-clause BSD license. Development on contiki has halted and focus moved to contiki-ng, which started as a fork and was first released on November 6th, 2017<sup>2</sup>. This new version does not offer modularization or loose coupling at the time this thesis was written, thus it is not used for further comparison.

As of version 3.0, Contiki's kernel includes code to load ELF binaries from its file system into dynamically allocated memory and to relocate them directly on the device. The whole ELF file needs to be transferred to the device and all additional processing required is done on the device. This approach differs from the work done in this thesis, as we aim to reduce the amount of data transferred to the device and to do as much processing as possible on the host machine. It does not include a mechanism to update components at run-time.

---

<sup>1</sup><https://github.com/contiki-os/contiki>

<sup>2</sup><https://github.com/contiki-ng/contiki-ng/releases/tag/release%2Fv4.0>

#### 3.1.2. Contiki-NG

Contiki-NG [22] is a fork of Contiki, which is still actively developed. The authors removed large portions of the original source code, including the dynamic ELF loader in 2017<sup>3</sup>. As of version 5.0, Contiki-NG does not support dynamic loading of code.

#### 3.1.3. FreeRTOS

FreeRTOS [1] is another real-time kernel for microcontrollers, developed as an open source project by Amazon. As of today, the only supported mode of operation is linking everything statically into one binary. Thus, no modularity is provided.

#### 3.1.4. RIOT OS

RIOT OS [2] targets a large number of IoT devices and supports a wide range of communication protocols. In version *Release-2024.10*<sup>4</sup>, its kernel does not support updating modules at run-time, a new image has to be built and flashed. It integrates Lua [10] as an optional feature, which could be used to implement a run-time update mechanism as described in 2.2.

#### 3.1.5. Zephyr Project

The Zephyr Project is an undertaking by the Linux Foundation and its community to build “[...] a best-in-class small, scalable, real-time operating system (RTOS) optimized for resource-constrained devices, across multiple architectures” [18]. At the time of writing, Zephyr does not support dynamically linking and loading modules out of the box. Users report building such functionality on their own using, for instance, *udynlink*<sup>5</sup> as a dynamic

---

<sup>3</sup><https://github.com/contiki-ng/contiki-ng/commit/6220aea14acbf2e21e9dbbd63da3a46f6025349a>

<sup>4</sup><https://github.com/RIOT-OS/RIOT/releases/tag/2024.10>

<sup>5</sup><https://github.com/bogdanm/udynlink>

linker for ARM-based MCUs on the device, similar to how contiki handled this.



## 4. Design and Concept

As mentioned in Section 2.3, the work in this thesis consists of two parts: changes to the kernel, and the design and implementation of a module server.

### 4.1. Device

One part of the implementation described in this thesis runs directly on the device. SmartOS supports multiple architectures, including the Aurix Tri-Core TC297TF and MSP430F5529. This implementation follows the existing code's intention of offering a high-level, platform independent API on top of lower level platform specific code. For this proof-of-concept implementation, the only supported chip is the MSP430. Its low level memory functions need to be designed first, as they define what the higher level API needs to support.

SmartOS is developed in C and uses a build system based on *make* [8]. This work was tested on an MSP430, more specifically the MSP430F5529, so SmartOS was compiled on GCC version 7.3.2.154, developed by Mitto Systems for Texas Instruments.

#### 4.1.1. Device Performance Classes

SmartOS categorizes devices into Device Performance Classes (DPC), based on their performance and resources. These classes are defined as follows in [3]:

- DPC-0 devices are very resource constrained. Keeping power consumption as low as possible is the main focus, as they, for instance might be battery powered. The device keeps track of installed modules, but does not perform any memory management tasks. The server has to store module and memory information and perform allocation and deallocation. Pluggability and interoperability checks are also performed on the server.
  - DPC-1 devices are still resource constrained, but they have their own Memory Manager (MeM), so they can perform the pluggability check. Furthermore, they keep a module's dependency list after installation. This information is necessary to perform the pluggability check when removing a module: the device must check if other modules depend on the one being removed. Upon updates, they receive a module's content and its dependency list.
  - DPC-2 devices have more processing power and memory than DPC-1, so they can also execute tasks that demand more memory, such as algorithms to relocate modules. Upon updates, they receive a relocatable ELF file already linked with global symbols, so it only needs to be relocated before being installed.
  - DPC-3 devices also offer more memory, so they store the symbol tables of installed modules. This gives them the ability to perform linking as well. Upon updates, they receive a relocatable ELF file, which needs to be linked and relocated before being installed.
  - DPC-4 devices have abundant memory and high processing power, so they can afford to store all the meta-information required by the Compatibility Check, and to execute the complex interoperability check. Upon updates, they receive a relocatable ELF file and its respective interoperability meta-information.
- All operations are executed at the device's side before installation. [...]

This work focusses on devices in class 1 (DPC-1) as in this class, both the device and the module server perform non-trivial tasks.



## 4.2. Module Server

A remote host machine is responsible for managing a library of modules, as well as building and transmitting them when requested by a device.

Each module is identified by a module specifier. This specifier is composed of two parts, the module's unique identification number and a version. This split allows both the server and the device to look for the same module in different versions, for instance to detect version changes. This identifier is called *Module Spec*. Figure 4.1 shows this structure, where the upper 8 bits are used for the identification number of a module and the lower 8 bits for its version.

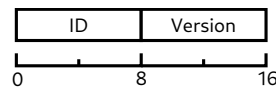


Figure 4.1.: Structure of a Module Spec

Version numbers start at 1. Version number 0 is special as it is used to designate "any version". This, for instance, is used to check if a module is installed on the device in any version or to request a module from the Module Server in the latest version.

To make it easier to port existing code bases and to ensure a flat learning-curve for the system the way code is written should not differ greatly from other systems. Especially the mechanisms of loose coupling should be abstracted in a way that makes them fully transparent.

At the time of writing, SmartOS supports three different architectures, MSP430F5529, TC297TF and RISC V. To be able to easily implement support for new architectures, or even programming languages used to write modules, the Module Server should be modular and adaptable. This is achieved by separating architecture- and compiler-specific code into so-called "backends". Each module is then able to specify which backend it is designed for.

The module server is written in Python 3. This language was chosen for its potentially high speed of development and rich ecosystem of modules useful for this work. One downside of Python is, that it is an interpreted

## 4. Design and Concept

---

language and thus inherently slower than compiled languages. As the module server is running on regular server or consumer hardware and not on low power devices, the lower execution speed is of no concern. One task of the module server is to analyze C code to automatically generate foreign module interfaces. To achieve this, the module server makes use of the clang project. clang implements a C/C++ frontend for the LLVM compiler framework [12]. It provides bindings [15] to its functions in Python, which can be used to parse any C program into an abstract syntax tree (AST). This simplifies automatic code analysis. As a memory optimization, the module server only transmits important parts of compiled modules to devices. See Section 4.3.1 for more details about this. To parse and extract data from ELF files, the module *pyelftools* [7] is used. Communication between module server and devices uses UART interfaces. *pySerial* [5] provides functions for accessing the serial port on multiple platforms.

To manage the installation of all dependencies *Pipenv* [25] is used. *pipenv* automates setting up Python projects by reading dependencies from a file and installing them in a so called virtual environment. The module server can then be launched from any supported operating system without further setup like so:

```
1 pipenv run python -m modserv --kernel <path/to/kernel.img> <serial device>
```

Listing 4.1.: Running Module Server from a command line

## 4.3. Update Protocol

### 4.3.1. Communication

The server and devices need a way to communicate with each other to be able to update modules. UART<sup>1</sup> is both easy to implement and supported in a large number of microchips. The microchip this implementation is tested on (MSP430F5529) has two UART interfaces [20]. On top of that, mature

---

<sup>1</sup>Universal Asynchronous Receiver Transmitter

libraries for Python are available. For these reasons, it is chosen as a first interface. In future work, other methods of communication can be examined. For instance, supporting wireless interfaces could offer OTA<sup>2</sup> installation and updates to modules.

To keep the amount of data exchanged as low as possible the host never sends a full ELF file. The ELF binary format is not designed for low-resource environments, as it can contain large string tables, redundant information and a significant overhead in its structure itself [21]. Therefore, only the relevant contents of the ELF are extracted and transmitted.

### 4.3.2. Synchronization

As the device will be communicating with a much faster server, synchronisation between the two parties is required. Otherwise, the server could overflow the device's UART buffer. To avoid this, the module server waits for the device to signal when it is ready to proceed to the next step of the protocol or receive the next batch of data.

## 4.4. Managing Memory

For any dynamic allocation the SmartOS kernel needs to be aware of memory boundaries and state. This is not a requirement in the current SmartOS kernel, as memory regions are handled by the linker and any additionally required memory by tasks is simply placed on the stack. When dynamically adding modules, the kernel needs to be able to allocate memory that outlives tasks. To allocate memory, it needs to know which memory regions are free and which are in use. As efficient memory management is a large topic and is not in the focus of this work, a basic implementation is provided which can be improved in future work.

---

<sup>2</sup>Over The Air

## 4. Design and Concept

---

As proposed in [3], a Memory Manager (MeM) is introduced. Its interface offers ways to allocate and write to memory, abstracting away any differences between ROM and RAM. Those differences arise from how ROM memory needs special treatment when writing.

To keep track of free memory, MeM stores pointers to the first unused address. Whenever memory is allocated, it moves those pointers forward by the amount of bytes requested. The data structure used to store this information is located in flash memory so it remains available throughout restarts of the device.

### 4.4.1. Flash Memory

As mentioned above, handling flash memory differs from RAM.

The MSP430F5529 offers 128 kilobytes of flash memory which is split into two memory banks with 64 kilobytes each. On top of that, there are four separate information and BSL<sup>3</sup> memory segments, each 128 bytes in size. The banks are partitioned into smaller segments, each 512 bytes in size. Since memory is mapped into a contiguous region on the MSP430 architecture, reading from flash memory does not differ from reading from RAM. Bank A, for instance, is mapped to the address 0x4400, Bank B to the address 0xC400.

Figure 4.2 depicts this memory layout for a device with four instead of two banks. Bank C and D only exist in models with larger flash memory.

Flash memory on the MSP430 can be programmed externally, for instance by using JTAG<sup>4</sup> or by the boot loader and CPU itself ([11]).

Section 5.5 goes into more detail about handling flash memory.

---

<sup>3</sup>Boot Strap Loader

<sup>4</sup>Joint Test Action Group, an interface commonly used for debug interfaces

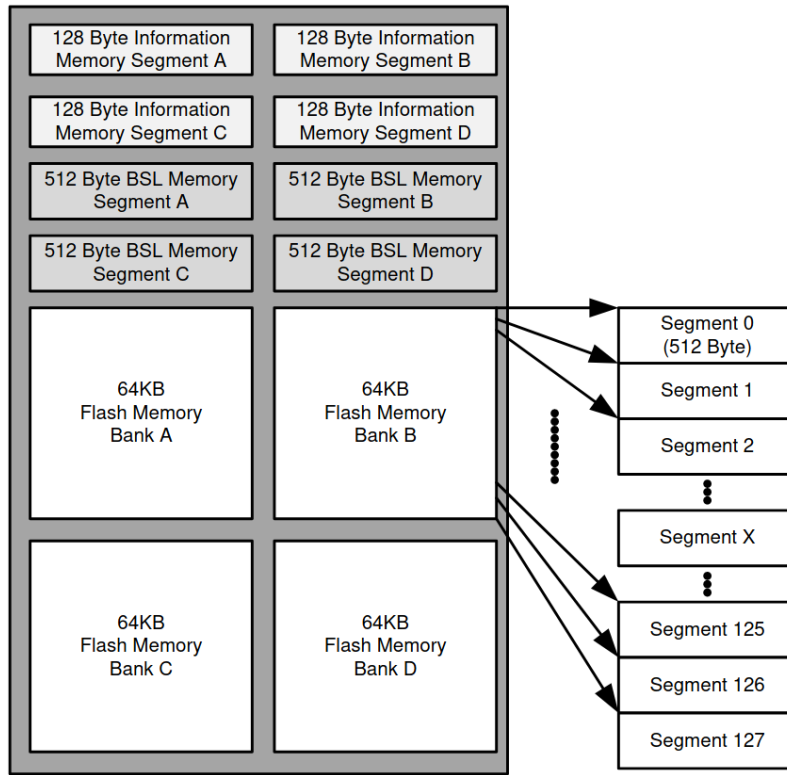


Figure 4.2.: Memory layout of an MSP430 with 256KB of flash memory ([20]).

## 4.5. Managing Modules

As modules are a dynamic building block of SmartOS, the kernel needs some methods to manage them. For instance, it needs to support adding and removing modules. This new responsibility is handled by the *Module Manager (MoM)*, as described in [3]. Its state is stored in a central data structure which is detailed in Table 4.2. The size of `LI[]` is defined at compile time and depends on the amount of ROM and RAM available. For the *MSP430F5529* used in this work it is set to possibly contain up to 32 modules. `virginLI`, `validLI`, `virginMod` and `emptyMod` are bit fields used to determine the state of modules on the device and to provide crash resiliency. See Section 4.5.1 for more information. `LI[]` (Load Information) is an array of module metadata required to load and start a module. Its structure is

## 4. Design and Concept

described in Section 5.3.

Field	Size [bit/module]	Description
virginLI	1	Load Information Slot is used/unused.
validLI	1	Load Information Slot is valid/invalid.
virginMod	1	Module is used/unused.
emptyMod	1	Module slot is empty.
LI[]	160	Module metadata array, see Table 5.4.

Table 4.2.: Module Manager data structure

### 4.5.1. Module & Load Information State

As detailed in [3], the status bit fields of the Module Manager are used to keep track of the installation state of modules. A value of 1 in `virginLI` denotes an unused Load Information slot in the `LI[]` array. As soon as the update protocol task allocates a slot for the new module, this bit is flipped to 0. Once the Load Information structure is written to flash memory, the corresponding bit in `virginMod` is set to 0. Finally, at the end of the installation process, the bit in `emptyMod` is set to 0.

Table 4.4 lists all possible states. The given bits represent values for `virginLI`, `virginMod`, `emptyMod` and `validLI`, in that order.

The memory layout of modules is illustrated in Figure 4.3. The left side represents the layout of two modules in RAM. Module A has two tasks, Module B just one. As can be seen, the code of both modules remains in ROM, next to their data for initialized, static variables. Somewhere earlier in ROM, in the core's memory region lies the Module Manager's data, storing state and Load Information of each module.

Configuration	State Description
1-1-1-1	Initial state, empty memory (only 1's).
0-1-1-1	Ongoing installation: writing to LI; else: installation failed while writing to LI.
0-0-1-1	Ongoing installation: writing to module ROM; else: installation failed while writing to module ROM.
0-0-0-1	Installed module.
0-0-0-0	Deleted module.
0-1-1-0	Corrupted LI: MoM detected failure during installation, and marked the LI as invalid.
0-0-1-0	Corrupted module: MoM detected failure during installation, and marked the LI as invalid.
Others	Not applicable.

Table 4.4.: Possible module state configurations [3]

## 4.6. Loose Coupling

As mentioned in Section 2.2, modules and other system components need to be decoupled to allow for easier code updates.

In a typical case, applications use libraries to avoid reimplementing functionality. For instance, functions for printing text messages are typically part of a systems core libraries. This avoids code duplication, decreases code size and helps sharing functionality between applications. There are two ways to link a library to an application, statically and dynamically.

The status quo of SmartOS is that everything is statically linked into one binary image. When statically linking a library with an application, its machine code is merged with the application's code at build time. On the one hand, this is less error prone and simpler to implement. The result is a standalone binary that can run on any of the devices it was built for. On the other hand this not only increases the size of the application binary, but also makes it impossible to share the library between multiple applications. Each application has to include its own version of the library. This results in

#### 4. Design and Concept

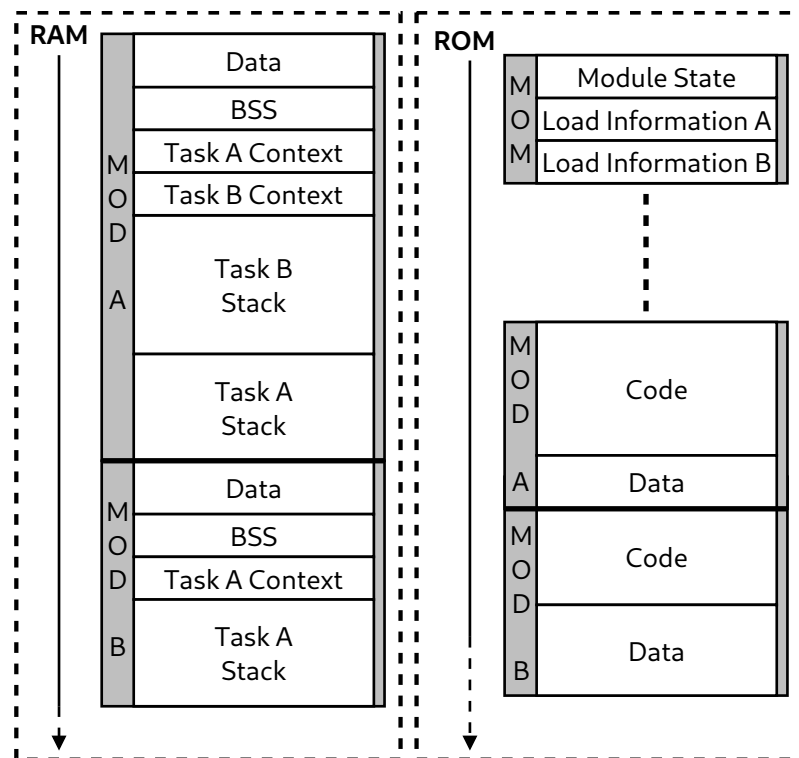


Figure 4.3.: RAM and ROM memory layout of modules

larger binaries, which can be an issue especially in the context of memory constrained environments like embedded devices.

Simplified, this is what happens when two binary objects are statically linked together:

- Append the text and data sections of one object to the other.
- Update references to symbols of object 2 in object 1 to point to the correct and likely relative addresses.
- Load the whole binary into memory.

Dynamically linked binaries, on the other hand do not contain the machine code of the libraries they use. They just contain enough information for the system to be able to look up those libraries and modify the binary's code to change function calls to the correct addresses, often at run-time. This process is called dynamic linking. The process can be done at build



or installation time, the first time the binary runs, or even on demand, just before a function is called. This makes binaries more lightweight and also allows two binaries to share the same library in memory.

Sharing code also introduces some new challenges: global variables in the scope of the shared code are then also shared between all users of the library. This can lead to unexpected behavior, as one application might change a variable that another one relies on. To mitigate this without restricting how libraries can be implemented (e.g. by forbidding global variables), the system needs to ensure that each task using the library has its own copy of the variables in memory. Some variables on the other hand need to be shared between tasks, for instance locks like mutexes. Libraries might need to make sure that only one application can access certain code regions at a time, so simply copying the lock for each application will actually break the intended behavior. Resolving these issues is a complex task and is not in the focus of this work.

There are several ways to achieve loose-coupling between system components. The approach chosen for this work focusses on transparency for module developers and a balance of performance between running and updating loosely-coupled code.

Module developers should not be forced to write code any different than if it were directly linked into the SmartOS core. That means, loosely-coupled function calls should not be distinguishable from calls to, for instance, the module's own functions. To achieve this, the abstraction that necessarily has to happen is hidden by automatically generating code wrapping the complicated parts. The Module Server requires a priori knowledge about modules and their exported functions, like return values and parameters. It uses this knowledge to generate jump tables and trampolines, as detailed in Section 5.4.1.

According to Peter Ruckebusch et al. [24], there are several approaches to updating code:

1. Replacing the whole image on the device.
2. Using scripting language interpreters.
3. Using specialized virtual machines

#### 4. Design and Concept

---

4. Splitting the firmware into smaller modules that can be swapped out at runtime.

Approach 1 describes what was mentioned before - a statically linked binary image, containing all components. Updating in this case means completely flashing the device.

Approach 2 introduces the overhead of storing an interpreter and scripts on the device's memory. This has significant impacts on performance and is likely not feasible for use in the context of resource-constraint embedded real-time operating systems.

Approach 3 suggests running byte code in a virtual machine. There are several projects like Maté [13], Darjeeling [4], MicroPython [9], Lua [10] or the version adapted specifically for embedded devices, eLua [16], which shows that there are both demand and use cases for this approach. This method entails some overhead, both in performance and memory as byte code has to be interpreted at run time. Maté for instance requires about 8 kilobytes [13] of memory. In benchmarks, the authors of Darjeeling report a 30 to 78x performance decrease compared to a native C solution [4].

For the following, Figure 4.4 shows how this thesis divides SmartOS into two layers: the monolithic kernel and the modular system. *Kernel* designates the monolithic part of the operating system and consists of tightly coupled components, for instance the memory and task manager as well as selected drivers. These components are not meant to be updated during run-time, thus they do not implement any of the loose coupling logic described.

The other part, the *system*, consists of tasks, libraries and drivers that can be updated, installed or removed.

Since embedded devices usually offer limited memory, sharing common code is an important feature. In monolithic kernels, the position of libraries in the binary are known to the linker and it can simply relocate function calls in components that use any of those libraries. In a modular system, the position of a component is not known until it is installed on the device and, on top of that, can change when it is updated.

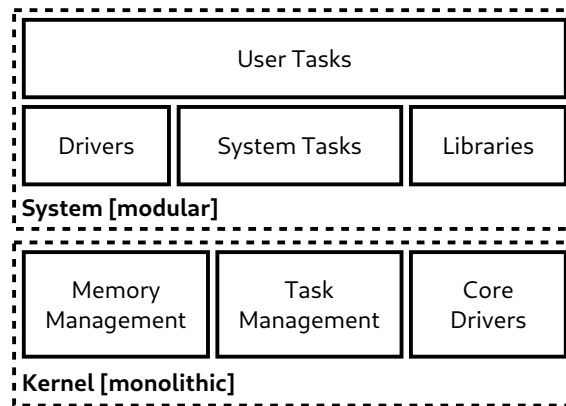


Figure 4.4.: Layers of the SmartOS operating system in regards to coupling.

This means that applications need to know the location of the drivers they use in the device's memory, as well as the locations of their functions. There are two scenarios that need to be covered:

- Scenario 1: An application is installed after the library it depends on. In this scenario, the library is already set up and working on the device. The kernel needs to report its location to the linker, so it can relocate all function calls in the application.
- Scenario 2: A library is updated and an application which depends on it is running on the system. This case is more complex, as the function addresses of the library might change. The update process needs to make sure that the application does not jump to obsolete addresses and thus probably executes invalid code.

Additional challenges arise in either scenario: if an application is updated while it currently is executing a function of a library, return calls may jump to an obsolete address. In a race condition, the application can also call a function of the library right before references to it are updated. It will then jump to the old address, which may be in a memory region that was already deleted or overwritten. Structure or layout of state of applications or libraries can also change between updates, for instance global variable types or members in structs were changed. The update process needs to make sure that existing state in memory is transformed to the new layout.

#### 4. Design and Concept

---

Also, if an application holds a lock on a resource like an interface the update process needs to make sure that the application releases the lock before the update is performed. Otherwise the update process would be blocked until the application releases the lock.

These problems are very complex and can be solved in many ways. This work will not focus on solving these issues and will therefore assume that applications are stopped before being updated. This makes sure that there are no on-going function calls as well as that applications will start with a fresh state.

There are several ways to couple components, offering different degrees of flexibility:

1. Directly link function addresses into applications. Use relocation to hard-code the targets of jumps to the location of the driver's functions. The exact addresses are known after installing a driver on the device and can be used to link the application. This way applications would work as if the driver was statically linked. There are no indirections on function calls, so this offers the highest performance and lowest memory usage. On the other hand, when a driver is updated and one or more function addresses change, the device would have to update all references in all installed applications - otherwise calls would end up in invalid memory regions, causing crashes. To be able to do that, the device would need to store a list of those references, for instance in the form of a relocation table as is present in ELF binaries (Figure 4.5). If an application holds pointers to functions then these will be invalid as well. Updating these pointers is not trivial.
2. Support a fixed set of libraries. This could be drivers and utility libraries. The addresses of these libraries are stored in a global trampoline table at a fixed address in the system's memory. When updating a library, only the address entry in the trampoline table needs to be updated. The downside is, that only these predefined libraries are supported and new libraries can't be installed on demand as that would likely require updating all applications (Figure 4.7).
3. Implement *trampoline* functions for each driver. As the driver knows the relative positions of its functions, it could offer an intermediate step that resolves a given identifier to a function's address. These

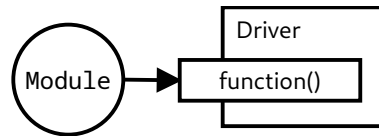


Figure 4.5.: Direct linking

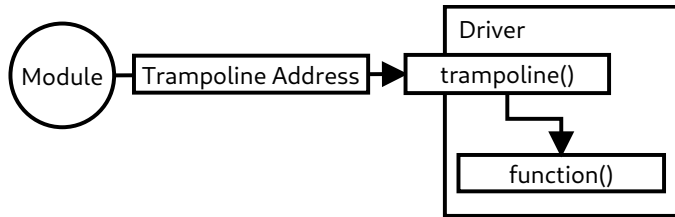


Figure 4.6.: Trampoline functions

functions help jump to other functions, they *bounce* a call, hence the name *trampoline*. This offers more flexibility, as the device now only has to update a single reference for each application. This introduces an indirection on function calls and slightly increases memory usage as the addresses of trampoline functions need to be stored somewhere. This could, for instance, be a global array in the kernel (Figure 4.6).

4. Function calls can be made fully dynamic by letting applications link against proxy functions. These functions look up the address of a driver in the system by an identifier at run-time and then call the driver's trampoline to get and return the real function's address. This offers full flexibility at the cost of performance. The kernel maps identifiers to the driver's trampolines and offers interfaces to retrieve its address. When updating a driver, only this reference has to be updated, no application needs updating. Not only does this introduce several indirections, it also includes a context switch from user to kernel space, which can be quite expensive (Figure 4.8).

These methods all assume that the modules are written in a language that supports direct linking and is able to call C functions by supporting the C FFI <sup>5</sup>.

The implementation will focus on the trampoline method, as it offers a good

---

<sup>5</sup>Foreign Function Interface

#### 4. Design and Concept

---

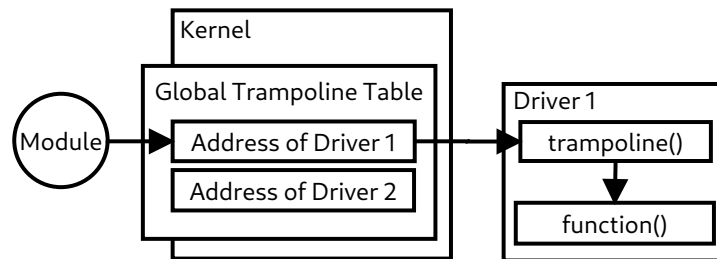


Figure 4.7.: Statically defined libraries

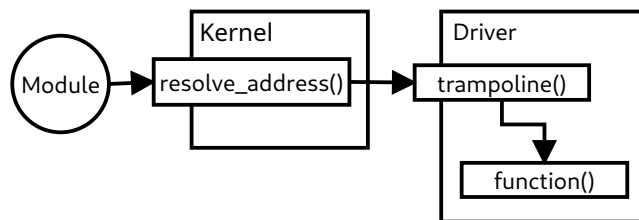


Figure 4.8.: Full dynamic resolution

balance between performance and flexibility. Each module has a trampoline table in its memory, populated with the addresses of the library functions it uses. This uses more memory than storing a single global trampoline table in the kernel memory, but it simplifies the update process as updates of one application will not affect others. This method requires the module server to know a module's dependencies at compile-time.

Summarizing, fully implementing code sharing and loose coupling in a system like SmartOS is a complex undertaking and goes beyond the scope of this work. This thesis will focus on the update process and will implement only basic functions of loose coupling.

## 5. Implementation and Realization

This section details the implementation of the module server, the update protocol, and the changes made to the SmartOS kernel.

Two modules are used to illustrate the principle of this implementation: `led_app` and `led_driver`. The latter implements functions necessary to control the LEDs provided on the MSP430 development board, whereas the former makes use of these functions to let these LEDs blink at a certain frequency. The full source code of both modules is shown in Listing B.2 and Listing B.3 respectively.

Albeit short, these modules demonstrate important features implemented in this work: A fully transparent way to use other, loosely-coupled modules, the possibility to use kernel functions and writing modules with and without defining tasks, module dependencies and installing modules at run time.

### 5.1. Module Server

As mentioned in Section 4.2, the Module Server manages, builds and transmits modules to devices. It is written in Python and uses *pyserial* to communicate with devices over a serial interface. Listing 5.1 shows the file structure of the server's code base. To manage dependencies, such as the aforementioned *pyserial*, *pipenv*<sup>1</sup> is used.

The main execution logic of the server can be found in *modserv/server.py*, more precisely in the function *run*. This function reads bytes from the UART interface connected to a device and waits for a module request sent from a device. More details about this request are described in Section 5.4.

---


















<sup>1</sup>Python package manager, <https://pipenv.pypa.io>

## 5. Implementation and Realization

---



The module compilation logic is abstracted and separated into so-called *backends*. This facilitates extending the server with support for different compilers. The base abstraction for backends consists of methods for analyzing a module's source code, to find out which functions need to be exposed to other modules, generating and compiling code and finally linking objects to get a finished module ready for transmission to the device.

Table 5.1.: File structure of Module Server.

File	Description
 linker	
 default.ld	Linker script used by the gcc backend.
 kernel_symbols.ld	Automatically generated linker symbols of the kernel.
 modserv	
 binary.py	Read addresses and section sizes from binary objects.
 build_config.py	Read & parse build.toml files.
 errors.py	Defines several error types.
 main.py	Main entry function of the Module Server.
 module.py	Defines a class that represents a module.
 server.py	Main Module Server and update protocol implementation.
 utils.py	Small utility functions and classes.
 backend	
 base.py	Abstract base class for backend compilation implementations.
 function.py	Utility class to store metadata of exported functions.
 gcc.py	Implementation of a gcc-based backend.
 modules	Contains all modules known to the server.
 led_app	See Figure 5.1 for the contents of a module folder.



 ...

 Pipfile	List of Python dependencies to install with the <i>pipenv</i> tool.
 run_server.sh	Shell script to start the Module Server.

---

Included in this implementation is a backend for gcc. As mentioned in Section 4.2, it uses clang to analyze source code. This library allows the backend to parse source files of a module into an abstract syntax tree (AST), which is then walked to iterate over all top level function definitions. For each function the backend checks if it is annotated using a so called *brief comment*, a comment right above the function header which starts with three slashes. This comment must contain a unique, numeric id, which is used in the trampoline to identify functions dynamically. Functions given such an id are considered to be *exported* and thus made available to other modules.

The list of exported functions is used to generate the module's trampoline function by constructing a C *switch* statement with one branch for each function.

### 5.1.1. Modules

All modules under the server's control are located in the *modules* folder. The general file structure of a module is shown in Figure 5.1.

Besides its source files, each module has to contain a file called *build.toml*. This file holds meta information about the dependencies required to build and distribute a module. Listing 5.1 shows this file for the module *led\_app*. Next to basic information such as a descriptive name and the author of the module, its ID, version, required operating system version, and available architectures are defined.

As mentioned in Section 4.2, different backends are supported by the Module Server. In this example, the code of all modules is written in C and can be compiled using *gcc*. Depending on the backend, it is possible to supply extra information, such as additional flags for the compiler or linker run.

## 5. Implementation and Realization

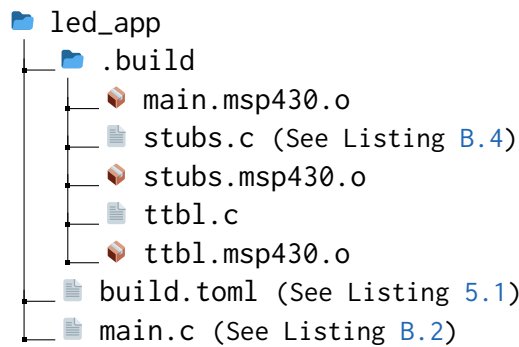


Figure 5.1.: Folder structure of the *led\_app* module.

```
1  [module]
2  name = "LED App"
3  author = "Fabian Schlager"
4  id = 0xC0
5  version = 0x01
6  os_version = 0x01
7  architectures = [ "MSP430" ]
8  backend = "gcc"
9
10 [dependencies]
11 led_driver = 0x01
12
13 [backend.MSP430]
14 cc_flags = [ "-g0" ]
15 ld_flags = [ ]
```

Listing 5.1.: `build.toml` of the *led\_app* module

Lastly, all dependencies and their required version are listed.

## 5.2. Memory Management

As stated in Section 4.4, dynamically managing memory is a prerequisite for any memory operations on devices during run-time.

The proposed Memory Manager (MeM) manages both ROM and RAM. It needs to know where unused memory on the device starts and ends so it can make sure to not overwrite data or code. This information is stored

in the constants `__localro_start`, `__localro_end` for ROM and `__localdata_start`, `__localdata_max` for RAM, which are defined when linking the kernel.

On the first start of the kernel on a device, MeM writes its full data structure, which can be seen in Table 5.2 to the flash memory. It consists of 2 address value pairs which point to the start of the free memory region as well as the end of the last allocated memory. On all subsequent reboots, it uses the values from flash memory for further operations. When memory is allocated, for instance by calling `mem_allocRAM(32)`, the value in `ram_current` is temporarily stored and then increased by the size of the allocation. After storing the new value to flash, the temporary value, which points to the start of the new allocation is returned.

Figure 5.2 visualizes the address value pairs in memory, after one allocation of RAM has happened. Reserved RAM and reserved ROM refer to memory occupied by the kernel itself. The variable `ram_current` points to the end of the allocated memory. As no allocations of ROM have occurred, `rom_current` is still equal to `rom_start`.

Pointer	Description
<code>rom_start</code>	Start of free ROM region
<code>rom_current</code>	Current end of dynamically allocated ROM
<code>ram_start</code>	Start of free RAM region
<code>ram_current</code>	Current end of dynamically allocated RAM

Table 5.2.: Memory Manager Data Structure

On top of functions to allocate memory, MeM offers two functions to write data to memory, abstracting away the differences between RAM and flash write operations: `mem_writeWord` and `mem_writeBuffer`. These functions include a small optimization: MeM assumes all non-reserved flash memory to be erased, enabling it to write without previously erasing. Every time data is written to flash memory using one of the utility functions mentioned, MeM stores the address of the end of that data. Memory before that address is considered *dirty*, thus requiring to erase flash before writing to it. Memory after that address can still be written to without erasing.

## 5. Implementation and Realization

---

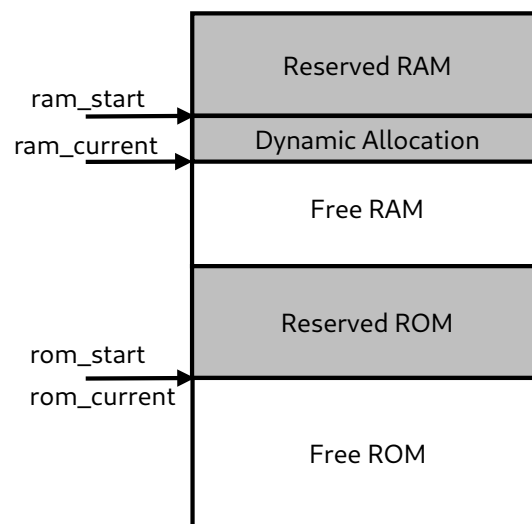


Figure 5.2.: Memory Manager data structure values

Field	Description
moduleSpec	Module Specifier.
tasks	Number of tasks defined in this module.
Trampoline	Address of the trampoline function.
TTBL	Address of the TTBL. See Section 4.6 for more information.
Base_ROM	Base ROM address.
Base_RAM	Base RAM address.
Size_TEXT	Size of the text section (executable code).
Size_DATA	Size of the data section (static initialized symbols).
Size_BSS	Size of the BSS section (static uninitialized symbols).
Size_Stack	Required stack size.

Table 5.4.: Load Information data structure

## 5.3. Module Management

Section 4.5 introduces the concept of a Module Manager (MoM). Table 5.4 shows the Load Information structure, which is the data necessary to load a module once it is installed on the device.

A data structure called `__moduleManager` contains the bit fields mentioned in Section 4.5 as well as an array of empty Load Information structs. Each index into this array is called a *module slot*. The size of this array as well as the size of the bitfields determines how many modules can be installed on the device simultaneously.

This data must not be lost upon restarts of the device, which is why it is persisted to flash memory. As an optimization, the bit fields mentioned above are initialized to a high state, considering that switching flash memory from low to high state requires erasing the memory beforehand: “The logical value of an erased flash memory bit is 1. Each bit can be programmed from 1 to 0 individually, but to reprogram from 0 to 1 requires an erase cycle.” [11].

The Module Manager offers an interface to register, search, update and load modules. Using `mom_allocModule()`, a new slot for a module can

## 5. Implementation and Realization

---

be allocated. This function simply iterates all available slots and returns the first unused one, or an invalid slot in case all slots are occupied. `mom_updateTrampolineAddress(old_address, new_address)` iterates all installed modules and, if a module has a trampoline table and contains the old address, replaces it with the new address. This function is used when updating a module, whose exported functions are used in other modules.

When the operating system starts up, it calls `mom_startup()`. This function iterates over all valid module slots and loads each module by calling `mom_loadModule()`, which initializes the module's RAM region by copying its data and BSS sections to memory and finally starts all tasks defined in the module.

### 5.4. Update Protocol

The protocol consists of several primitive data types used to exchange data between device and Module Server: 8 bit (byte), 16 bit (short) and arbitrary length byte arrays. The device prefixes its data with control bytes to indicate what kind of data follows, as described in Table 5.6.

Accompanying the following explanation is an extensive sequence diagram. To improve the readability, this diagram is split into multiple parts. Notable operations are indexed sequentially throughout these parts. The following text references those operations by using this notation: "Fig. 5.4a, 22", which points to Figure 5.4a, annotation 22.

In this implementation, the device starts the installation or update process by sending a request for a specific module to the server. As mentioned in Section 4.3.2, one issue that is addressed is synchronization between device and server in the UART interface. Both implementations use ACK signals to acknowledge the successful transmission of values or to indicate having finished an operation. This is necessary to avoid causing buffer congestion on the device, for instance when the server is sending a lot of data or needs to wait for the device to finish some operations before continues to send more data.

Control Byte (Decimal Value)	Payload	Description
REQUEST_MODULE (128)	Module ID (Short), Architecture (Byte), DPC (Byte), OS Version (Short)	Requests a module <i>Module Id</i> from the server.
SEND_BYTE (129)	Byte	Sends a byte to the server.
SEND_SHORT (130)	Short	Sends a short to the server.
SEND_DBG (35, '#')	Newline terminated string	Sends a debug string to the server.
SEND_ACK (250)	<i>None</i>	Acknowledges the last transfer, lets the server know it can continue.

Table 5.6.: List of update protocol commands

The following sections describe all parts of the update protocol in detail, following the flow of the installation of the aforementioned *led\_app*.

As described in section 4.1.1, depending on the class of a device, components are either linked on the module server or directly on the device. For devices in DPC 1, which this thesis focuses on, the device is managing its memory autonomously but modules are built and linked on the server.

A module can be an application, meaning that it defines one or more tasks that will be executed as soon as it is loaded on the device, or it can be a library, simply offering utility functions but defining no tasks.

Each module can depend on other modules, as demonstrated in the module *led\_app*, which depends on the module *led\_driver* to provide functionality to control the surface-mounted LEDs present on the used MSP430 evalua-

## 5. Implementation and Realization

---

tion kit<sup>2</sup>. Both modules need to be registered at the server that builds the application. The application specifies a list of dependencies.

In short, the process works as follows:

- Server: Preparation & Setup
- Device: Request a module from the server.
- Server: Verification, Compilation & Transmission: Verify that the module exists and is compatible with the device. Compile it and all of its dependencies for the device's architecture. Send all dependencies.
- Device: Dependency Installation: Install the dependencies and report back their location in memory to the server.
- Server: Relocation & Transmission: Relocate the trampoline address of each dependency in the module's TTBL according to their address. Send the module.
- Device: Module Installation: Install the module.

Each item in this list is described in detail in the following sections.

### 5.4.1. Preparation & Setup

As modules may use kernel functions, the linker needs to know their addresses at link time. Thus, as a first step, the Module Server extracts all symbols of type *function* or *object* which are not hidden nor start with a double underscore. The latter can be used to designate objects or functions that should not be exposed to modules. This is used by the Module Manager to hide internal structures as it is not desired to let modules circumvent the API and make changes to these structures directly, similar to access specifiers *public* and *private* in C++.

The function addresses are calculated relative to the section they appear in. These symbols and their sections are then written to a file formatted in a way the linker *ld* understands. An example of the result of this operation can be seen in Listing 5.2, defining the addresses of the *text* section and two functions `crc_8` and `update_crc_8`. Listing B.1 contains the implementation of this operation.

---

<sup>2</sup>MSP-EXP430F5529LP



```
1 ...  
2 __text_start = 0x440e;  
3 crc_8 = __text_start + 0x0000;  
4 update_crc_8 = __text_start + 0x001e;  
5 ...
```

Listing 5.2.: Example of extracted kernel symbols

After extracting the kernel's symbols, the server loads available modules from the *modules* directory by iterating its entries. Each subfolder represents one module, the structure of such folder is shown in Figure 5.1.

The following steps are then performed for each module:

- Module compilation: The module itself is compiled, but not yet linked.
- Trampoline generation: A trampoline function is generated, if necessary.
- Stub generation: Stubs are generated for each dependency.
- TTBL generation: A table of trampoline addresses (short: TTBL) is generated.

### Module Compilation

When a new module is registered, it will automatically be compiled to a relocatable, unlinked object as a preparation for upcoming requests. This has an advantage of reporting compilation errors as soon as the server starts as opposed to during communication with the device. This compiles the module for each of its supported architectures. As mentioned previously, only MSP430F5529 is supported at this time.

### Trampoline Generation

For each module the module server generates a function postfixed with *\_trampoline* (for instance: `led_driver_trampoline`), which takes a function ID as argument and returns a pointer to the function defined with this ID. The trampoline function of the LED driver module can be seen in Listing 5.5. If no functions are exported by a module, no trampoline is generated.

## 5. Implementation and Realization

---

### Stub Generation

In this step each dependencies' code is parsed to find out which functions the module wants to export to other modules.

The Module Server automatically generates stubs for each exported function. When linking a module to its dependencies, it is not linked to the original object but rather to its stubs. All stubs are written into a file called *stubs.c*.

As the original source code is parsed, the stubs can use correct types for return values and function parameters, preserving compile-time type safety. However, instead of just calling the original function, a look-up in the TTBL is performed. This table stores the address of the dependencies' trampoline function, which is called with the specific function's ID. The trampoline returns the address to the function itself. See Section 5.4.1 for more information.

Listing B.4 lists the stubs generated for the LED app module.

```
1 Result_t LEDsetState(led_t a0, ledState_t a1) {  
2     uint8_t* (*trampoline)(uint8_t) = TTBL[0];  
3     Result_t (*actual_function)(led_t, ledState_t) = trampoline(2);  
4     Result_t result = actual_function(a0, a1);  
5     return result;  
6 }
```

Listing 5.3.: Expanded stub LEDsetState

For better readability, Listing 5.3 rewrites the stub generated for `LEDsetState` to make it easier to understand. In line 2, the trampoline is looked up in the TTBL. In this example, *led\_driver* is the first dependency of *led\_app* and thus is also the first entry in the TTBL. As shown in Listing 5.4, `LEDsetState` has ID 2. This is the argument that is passed into `trampoline` in line 3, which then returns a pointer to the actual function defined in *led\_driver*. The arguments given to the stub are passed into the actual function in line 4 and its return value is returned in line 5.

Figure 5.3 illustrates this call flow, including the memory borders between *led\_app* and *led\_driver*.

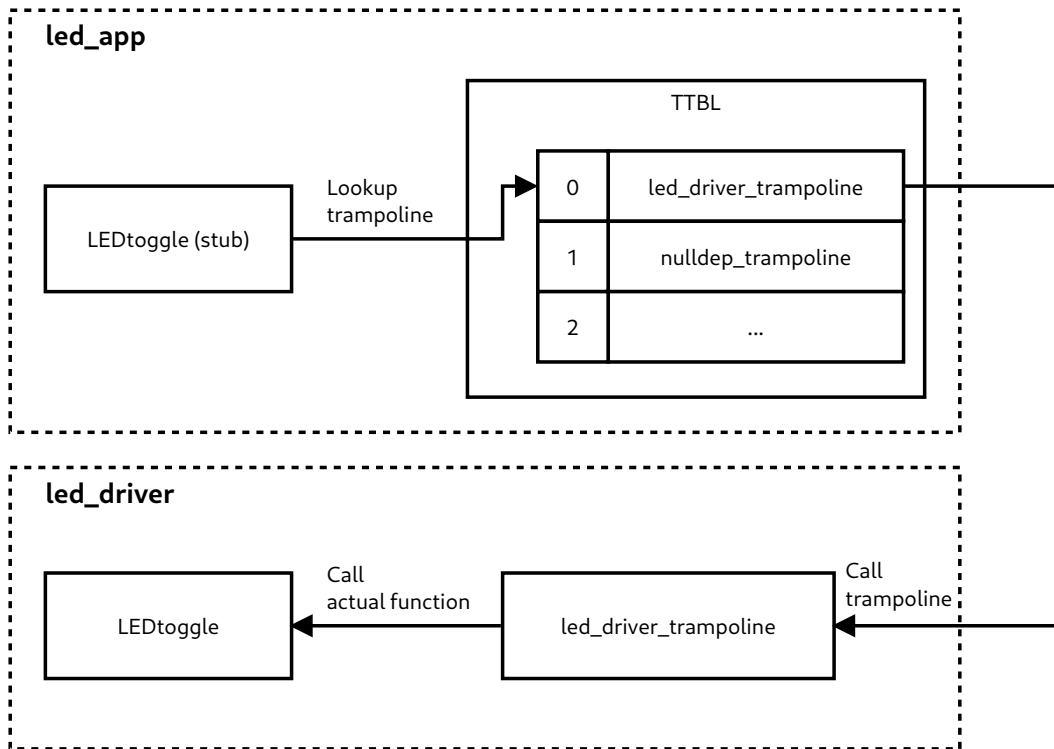


Figure 5.3.: Example call flow, from stub to actual function

Listing B.2 shows how these stubs can be used fully transparent, just like calling the original functions. This makes loosely-coupled function calls transparent for application developers.

Listing 5.4 shows three functions from the LED driver module. Note that they are annotated with a special, long-style comment, defining a unique ID for each function, for instance ID 2 for the function LEDsetState. This ID is used to identify this function in the trampoline and to generate stubs, as can be seen in Listing B.4.

### TTBL generation

The trampoline table contains the addresses of the trampolines of dependencies used in this module. For instance, the *led\_app*'s TTBL contains the

## 5. Implementation and Realization

---

```
1  /// ID: 0
2  Result_t LEDconfigure(led_t led) {
3      if(io_configureDirection((led >> 4), (led & 0x0F), 1) == FAILURE){
4          return FAILURE;
5      }
6
7      return io_configureSelection((led >> 4), (led & 0x0F), 0u);
8  }
9
10 /// ID: 1
11 Result_t LEDtoggle(led_t led) {
12     return io_togglePin((led >> 4), (led & 0x0F));
13 }
14
15 /// ID: 2
16 Result_t LEDsetState(led_t led, ledState_t state) {
17     return io_setPinValue((led >> 4), (led & 0x0F), state);
18 }
```

Listing 5.4.: Exported functions of `led_driver` module

address of the *led\_driver*'s trampoline.

As most of the parts before, the trampoline table is generated automatically as well. It is written to a file called *ttbl.c*. Listing 5.6 shows the TTBL generated for the LED app. The order is determined by the order the dependencies are listed in the module server. This is important, as it is also used in the generated stubs (see Section 5.4.1) as the index in the TTBL. The TTBL ends on a NULL, to allow the kernel to iterate over all entries.

### 5.4.2. Module Request

In the current implementation, the device starts the installation or update process by sending a request for a specific module to the server (Fig. 5.4a, 1).

This request contains all information needed for the server to look up the correct module. *Architecture* is an identifier to let the server know which platform it needs to build the module for. *DPC* denotes the way updates are handled. See Section 4.1.1 for more information. On top of that, the server needs to know which version of the core operating system is running on

```

1  #include <led_driver/led.h>
2  #include <ostypes.h>
3
4  uint8_t* __attribute__((used,section(".text"))) led_driver_trampoline(uint8_t id) {
5      switch (id) {
6          case 0: return (uint8_t*)&LEDconfigure;
7          case 1: return (uint8_t*)&LEDtoggle;
8          case 2: return (uint8_t*)&LEDsetState;
9      }
10 }

```

Listing 5.5.: Automatically generated trampoline function for led\_driver module

```

1  #include <ostypes.h>
2  #include <stddef.h>
3  uint8_t* led_driver_trampoline(uint8_t id);
4  const uint8_t* (*TTBL[])(uint8_t) __SECTION(".TTBL") = {
5      led_driver_trampoline,
6      NULL
7  };

```

Listing 5.6.: Automatically generated trampoline table (TTBL) for led\_app module

the device, to be able to link the correct symbols for this version. It looks up the module in its database (Fig. 5.4a, ②) and responds with one of the following status codes:

- **0:** The module was found and is available for this architecture and OS version, the process can continue.
- **1:** The module was not found, the process will abort.
- **2:** The module was found but is not available for this architecture.
- **3:** The module was found but is not available for this OS version.

Only a zero allows the device to continue the protocol, all other values are unresolvable errors, that must result in a cancellation (Fig. 5.4a, ④). How to further handle these errors is not covered in this work. Given that the lookup was successful, the server now responds with memory requirements of the module given by the sizes of its binary sections, in the following order:

1. Data

## 5. Implementation and Realization

---

2. BSS
3. Text
4. Stack

This is followed by the number of tasks defined in the module. All information the core needs to load a module at the start of the device is stored in a data structure called `LoadInformation_t`. Its composition is described in Table 5.4. The device creates such a structure and initializes it with the values it receives from the module server (Fig. 5.4a, ③). The structure is not complete at this point, the missing information, such as the address of the TTBL and base addresses in both ROM and RAM is added later on.

### 5.4.3. Dependency Installation

Each module can specify dependencies, which will be installed in this step, if necessary. In this example, *led\_driver* is a dependency of *led\_app*. For this, the server sends each dependency's module ID (Fig. 5.4b, ⑤). The device then determines if a dependency is already installed by trying to look it up in the kernel's module list (Fig. 5.4b, ⑥). If it could not be found, it will request this module from the server, starting a new module request process (Fig. 5.4b, ⑦). Otherwise, the device responds with the dependency's metadata and signals the module server to continue (Fig. 5.4b, ⑧) with the next dependency.

Verifying and installing dependencies is one part of the pluggability check of a module as described in [3].

### 5.4.4. Relocation & Transmission

The second part of the pluggability check is to verify, that there is enough memory available on the device. The total memory required is calculated like so:

$$Total_{ROM} = Size_{TEXT} + Size_{DATA}$$

$$Total_{RAM} = Size_{DATA} + Size_{BSS} + Size_{STACK} + Size_{TaskContext} * Number_{Tasks}$$

The device attempts to allocate the required space in ROM and RAM for the actual module (Fig. 5.4c, 9). If this is unsuccessful, the process is aborted.

It then determines if an older version of this module is already installed (Fig. 5.4c, 10). If this is the case, it will now disable the currently installed version (Fig. 5.4c, 11) by setting the specific bit in the Module Manager's state bit fields.

Now, that memory for all modules to be installed is allocated, the device sends back all addresses to the Module Server (Fig. 5.4d, 12, Fig. 5.4d, 13). It now has all information required to successfully link and relocate the module. It does so by merging the kernel's symbols generated in Section 5.4.1 by the trampolines of the dependencies. These symbols, together with the base addresses of the module are then used to relocate the object created in Section 5.4.1 and build a final ELF binary (Fig. 5.4d, 14).

At this point, the Module Server can transmit the missing information of the Load Information structure to the device, namely the addresses of the trampoline and the TTBL (Fig. 5.4d, 15). The device commits the structure to flash memory and marks it as valid in the Module Manager (Fig. 5.4d, 16).

Transmission of code then starts. The Module Server sends the actual length of the code. This is necessary as the code length might have changed after linking and relocating, for instance when absolute jumps are replaced by relative ones. Then the code is transmitted in chunks of 16 bytes (Fig. 5.4e, 18).

The device receives such a chunk, writes it to flash memory and then signals the Module Server to continue with the next block. At the same time both the device and the server calculate a CRC-8 checksum of all bytes transferred (Fig. 5.4e, 19 & 20). After the transmission is completed, this checksum is sent back to server, which either confirm its correctness or reports an error. In the latter case, the process is retried from step Fig. 5.4e, 18, otherwise it continues by transmitting the binary's data section in the same manner, which is omitted from the sequence diagram.

### 5.4.5. Module Installation & Loading

After a successful transmission the module slot is marked as occupied and the module is loaded. To load a module, the kernel starts by copying its data section to RAM, initializing the BSS portion to zero as well as reserving enough space for the task stacks and data structures required for context switches.

What this looks like in memory can be seen in Figure 4.3. Modules at rest lie in ROM (right-side), together with the Module Manager's information. When they are loaded, their structure in RAM looks like on the left side.

## 5.5. Storing in Flash Memory

Storing data in flash memory is not trivial, as the method of writing depends on the state the flash is in. When flashing the SmartOS kernel, the flash area is erased. Due to the flash memory's physical properties, in its erased state all bits are 1. Flipping a bit to 0 is trivial, but the reverse operation requires the flash to be erased again. Erasing memory on the MSP430F5529 only works in blocks of one segment, with a segment being 512 bytes long.

The smallest unit that can be written to flash is a byte (8 bit). Given that, to change a single bit from 1 to 0 the following steps are necessary:

- Read the byte from flash to RAM
- Modify the bit in RAM
- Write the byte back to flash

Changing a single bit from 0 to 1 is even more expensive. To do that, that bit needs to be erased. When erasing, instead of a byte, the smallest unit is a segment. As mentioned before a segment consists of 512 bytes. Considering that, the process of writing a 1 to flash consists of the following:

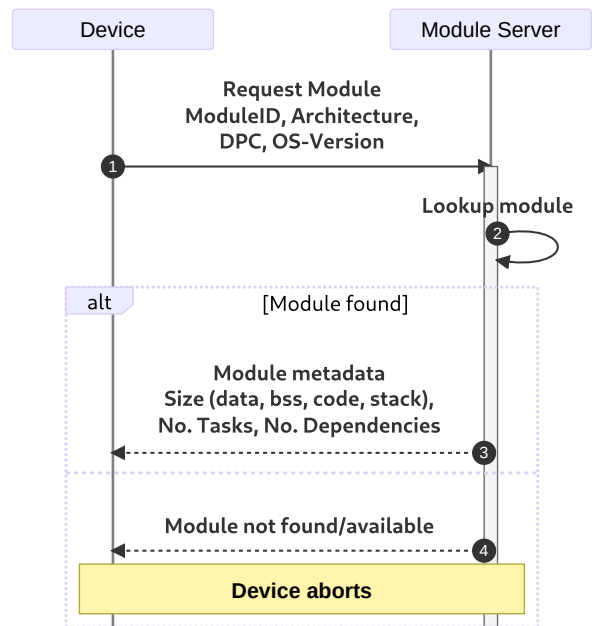
- Read the whole segment into 512 bytes of RAM
- Change the single bit in RAM
- Instruct the flash controller to erase the segment
- Write back the whole segment from RAM to flash



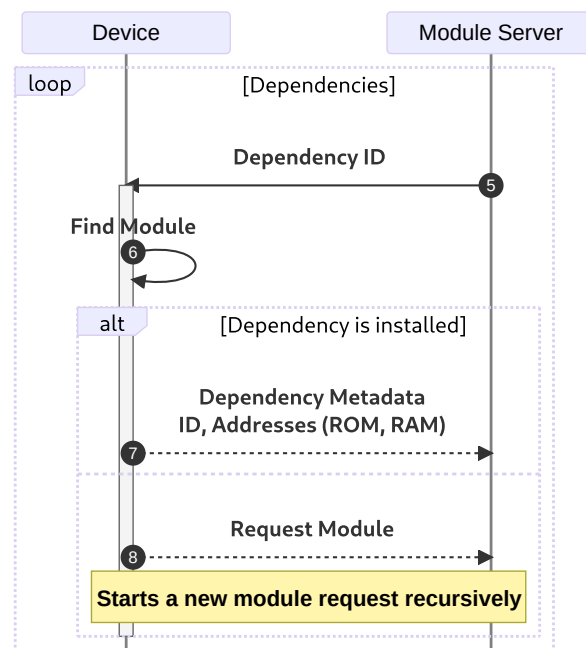
To avoid modification of unwanted areas in the memory, the whole segment needs to be loaded into RAM. Then, any bits can be modified as required. The affected flash segment is erased and then the full 512 bytes are written back. Special care has to be taken when modifying memory areas that span more than a single segment, as this process has to be repeated for each segment. A segment may currently be in use, as the CPU can run code directly from flash. This case is not covered in this implementation and needs special treatment.

To optimize the usage of flash memory, its state needs to be tracked by the kernel. In this implementation, this is the job of the Memory Manager. For instance, it could be desired behaviour to never erase flash as long as there is fresh memory available, as it is an expensive operation. When requesting memory in flash, MeM should check for an area that is large enough to hold the requested amount of memory and is still initialized to 1. Repeated memory allocation and deallocation can lead to fragmentation of the flash memory, reducing the amount of memory available for future allocations. Fixing this is not covered in this implementation.

## 5. Implementation and Realization

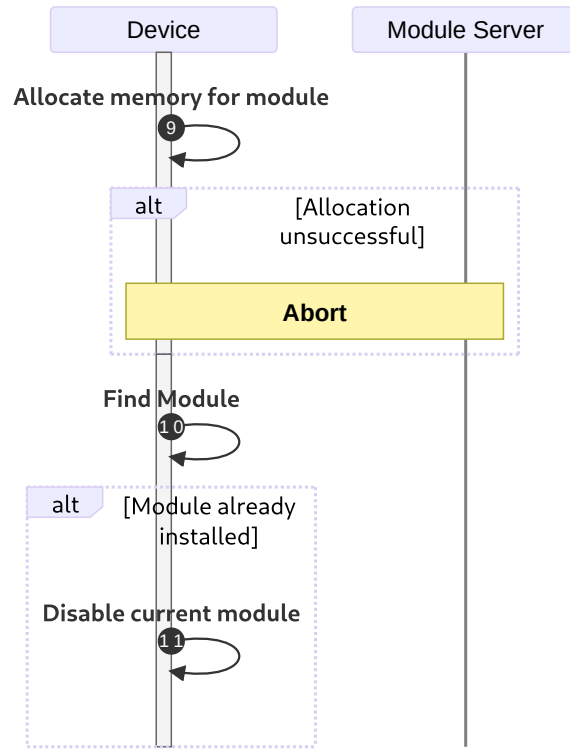


(a)

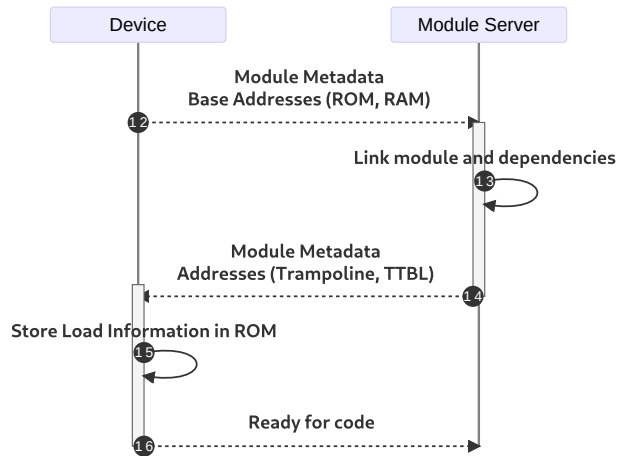


(b)

## 5.5. Storing in Flash Memory



(c)



(d)

## 5. Implementation and Realization

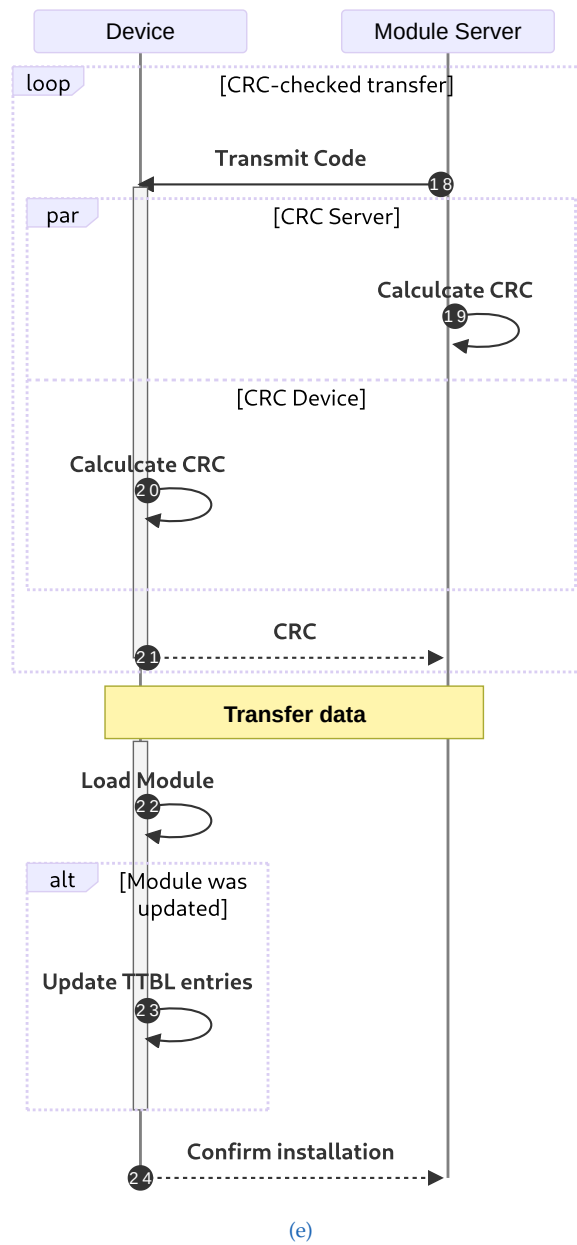


Figure 5.4.: Sequence diagram of update protocol flow

## 6. Evaluation and Analysis

The paper [3] this work builds upon provides analysis and evaluation of processing time and the amount of exchanged data of the proposed update protocol. The following repeats some of the measurements, as the implementation details have changed.

### 6.1. Size Analysis of a Module

As described in Section 4.3.1, only relevant sections of ELF files are transferred from the Module Server to the device. This amounts to the *text*, *data* and *rodata* sections. Figure 6.1 visualizes the ratio of a binary object file (stripped and not stripped) to just the relevant sections. The binary was built including debug information.

### 6.2. Performance Analysis

As described in Section 5.4.1 two layers of indirection are introduced to calling functions inside modules. In this section the performance cost of these indirections will be analyzed by looking at the machine code generated by *msp430-elf-gcc*. The machine code was disassembled using *msp430-elf-objdump*.

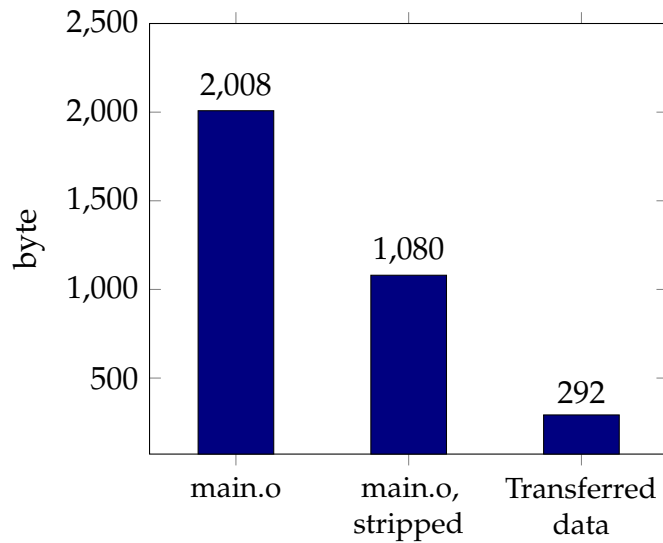


Figure 6.1.: Comparison of byte size of ELF file and actual data transmitted

### 6.2.1. Analysis of TTBL based calling

The call to the stub is shown in Listing 6.1. The relative address of the function `LEDtoggle` is moved into register 9, the parameter to the call (`LED_01`, 16) to R12. Finally, the function is called.

Listing 6.2 shows the disassembly of the auto-generated stub `LEDtoggle`. It begins with the contents of R10 being pushed to the stack, as required by the MSP430 calling convention (see manual [19]). The argument passed in to the stub is then stored in R10 for later use, as R12 is also required to make an intermediate call to the trampoline. Lines 7 to 9 call the trampoline of the LED driver module with argument 1. Notable is line 8, as this directly uses the TTBL without any further redirections. For comparison, listing 6.3 shows a function call to a module in the second slot of the TTBL. The access to the TTBL is optimized by directly accessing memory at the correct offset.

The function address returned by the trampoline is stored to register 13, the argument previously stored in register 10 is moved back to register 12 and the actual function is being called in line 12. At the end, register 10 is

restored according to the calling convention and the stub returns back to its caller.

As a comparison, Listing 6.4 shows the disassembled function call to `LEDtoggle` without loose coupling. Section 6.2.2 discusses the overhead of loose coupling in detail.

```
1 ; LEDtoggle(LED_01);
2
3 mov #-14250, r9 ; Set address of stub
4 mov.b #16, r12 ; Set first argument
5 call r9 ; Call stub
```

Listing 6.1.: Calling the LEDtoggle stub

## 6. Evaluation and Analysis

---

```
1 ; Result_t LEDtoggle(led_t a0) {
2 ;   return ((Result_t (*)(led_t))(TTBL[0](1)))(a0);
3 ; }
4
5 pushm #1, r10 ; Save r10 to stack
6 mov r12, r10 ; Save r12 (first argument) in r10
7 mov.b #1, r12 ; Use 1 as argument of trampoline function
8 mov 0xc770, r13 ; 0xc770 is TTBL[0]
9 call r13 ; Call trampoline function
10 mov r12, r13 ; Move the returned value to r13
11 mov r10, r12 ; Restore r12 from r10
12 call r13 ; Call actual function with r12 as argument
13 popm #1, r10 ; Restore r10 from stack
14 ret ; Jump back to caller
```

Listing 6.2.: Generated assembly code for *LEDtoggle* stub

```
1 ; void null_func() {
2 ;   ((void (*)(()))(TTBL[1](0)))();
3 ; }
4
5 clr.b r12 ; Clear r12
6 mov 0xc772, r13 ; 0xc772 is TTBL[2]
7 call r13 ; Call trampoline function
8 call r12 ; Call actual function
9 ret ; Jump back to caller
```

Listing 6.3.: Generated assembly code for *null\_func* stub

```
1 ; LEDtoggle(LED_01);
2
3 mov #19702, r8 ; #0x4cf6 is the address of LEDtoggle
4 mov.b #16, r12 ; #0x0010 is the parameter
5 call r8 ; Call LEDtoggle
```

Listing 6.4.: Calling the *LEDtoggle* function directly



### 6.2.2. Clock cycle analysis

In this section, the generated assembly code is analyzed to calculate the total number of CPU clock cycles used in each method. With this number, the differences in performance can be compared on the MSP platform. Other chipsets, or more specifically other compilers, might generate different code.

Quoting the "MSP430 User's Guide": "The number of CPU clock cycles required for an instruction depends on the instruction format and the addressing modes used – not the instruction itself." [11]. MSP430 makes a distinction between single- and double-operand instructions. Single-operand instructions operate on one argument, double-operand ones on two. On top of that, the arguments matter. Figure 6.2 lists the clock cycles necessary for each addressing mode for single-operand instructions.

The following text uses abbreviations for addressing modes and instruction types listed in Table 6.1.

Name	Description
SO	<b>Single-operand.</b> An instruction taking a single argument.
DO	<b>Double-operand.</b> An instruction taking two arguments.
IA	<b>Immediate Access.</b> Constants which are included in the encoded instruction.
R	<b>Register.</b> Directly accessing register contents.
AA	<b>Absolute Addressing.</b> Accessing an absolute memory address.

Table 6.1.: Abbreviations for addressing modes and instruction types

Table 6.2 analyses the method of directly calling a function, as is also shown in listing 6.4. The line numbers match those in the listing.

Table 6.3 analyses the modular method of calling a function, as is also shown in Listings 6.1 and 6.2. As the first part of calling the stub function is the exact same as calling the function directly, it will be skipped in the analysis. The line numbers match those in the respective listings.

## 6. Evaluation and Analysis

Line	Instruction	Type	Cycles
3	mov #19702, r8	DO, IA, R	2
4	mov.b #16, r12	DO, IA, R	2
5	call r8	SO, R	4
Total:			8

Table 6.2.: Clock Cycle Analysis of direct function call

Line	Instruction	Type	Cycles
4-5	-	See table 6.2.	8
5	pushm #1, r10	DO, IA, R	2
6	mov r12, r10	DO, IA, R	1
7	mov.b #1, r12	DO, IA, R	2
8	mov &0xc770, r13	DO, AA, R	3
9	call r13	SO, R	4
10	mov r12, r13	DO, R, R	1
11	mov r10, r12	DO, R, R	1
12	call r13	SO, R	4
13	popm #1, r10	DO, IA, R	2
13	ret	SO	4
Total:			32

Table 6.3.: Clock Cycle Analysis of indirect function call.

As, Ad	Addressing Mode	Syntax	Description
00, 0	Register	Rn	Register contents are operand.
01, 1	Indexed	X(Rn)	(Rn + X) points to the operand. X is stored in the next word, or stored in combination of the preceding extension word and the next word.
01, 1	Symbolic	ADDR	(PC + X) points to the operand. X is stored in the next word, or stored in combination of the preceding extension word and the next word. Indexed mode X(PC) is used.
01, 1	Absolute	&ADDR	The word following the instruction contains the absolute address. X is stored in the next word, or stored in combination of the preceding extension word and the next word. Indexed mode X(SR) is used.
10, –	Indirect Register	@Rn	Rn is used as a pointer to the operand.
11, –	Indirect Autoincrement	@Rn+	Rn is used as a pointer to the operand. Rn is incremented afterwards by 1 for .B instructions, by 2 for .W instructions, and by 4 for .A instructions.
11, –	Immediate	#N	N is stored in the next word, or stored in combination of the preceding extension word and the next word. Indirect autoincrement mode @PC+ is used.

Figure 6.2.: Source and destination addressing in MSP430 instruction set [11]

Comparing the results of both CPU clock cycle analyses it can be seen, that the direct method takes 8 cycles, whereas the modular approach takes 32 cycles. This means that each function call to a different module in the modular approach takes four times as many cycles as a direct call.

Test	Time [ms]
Modular installation	7800
... Module Requirements	533
... Dependency Installation	3060
... Memory Allocation	70
... Module Allocation	295
... Code Transmission	3540
... Module Loading	0.23
Flash monolithic image to first start of task	$8777 + 176 = 8953$
Boot up and start task (Install app, start measure, reboot)	68.6
Monolithic Boot up and start task	68.2

Table 6.4.: Time measurement results

### 6.3. Time Analysis

In this section, the run time of several parts of the implementation is measured and compared to a monolithic version. The monolithic version does not contain the update protocol task but directly builds and runs the task defined in *led\_app*. It does contain all other functionality added to the kernel, but as this is never called it should not affect measurements.

Most measurements were performed by turning one of the GPIO pins of the MSP430F5529 to high (applying 3.3 Volt), to signal the start of a test. Turning the pin to low (0 Volt) again signifies the end of the test. An oscilloscope (Rigol DS1054) is then used to trigger on these signals and measure the time. Figure 6.3 shows an example of this method.

For these measurements all debug output over UART originating from the device is disabled.

Table 6.4 lists the results of the tests, the following describes each test in more detail. Transmission of data is not included as *led\_app* does not contain any data. This process is the same as sending code, so this has no impact on this evaluation.

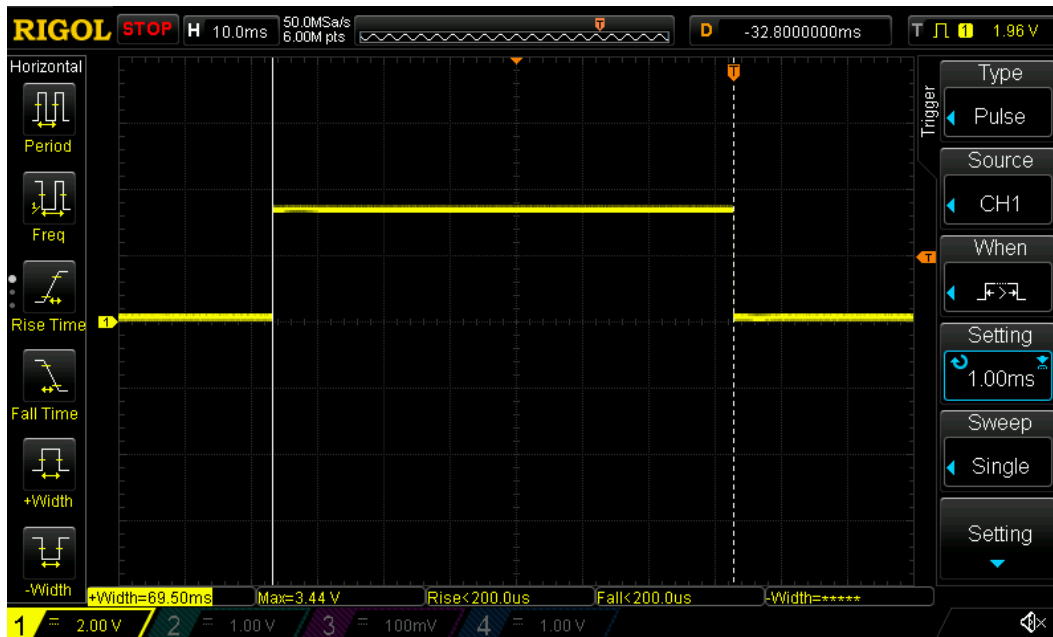


Figure 6.3.: Time measurement performed on Rigol DS1054

### 6.3.1. Modular Installation

The whole process of installing a module according to the implementation done in this work takes about 7800 milliseconds.

The following splits this up into smaller steps. The time they take does not add up to the full 7.8 seconds, 302 milliseconds are missing. This time is most likely spent on writing the Load Information to flash the second time as well as smaller operations in between the ones measured.

Figure 6.4 visualizes the individual portions.

- **Modular installation:** This test performs the process detailed in Section 5.4, which is installing *led\_app* on a system that does not contain any modules yet. This includes installing the *led\_driver* dependency and finishes as soon as the application starts. The following tests dissect this process further to show which parts make up the most of the time required.

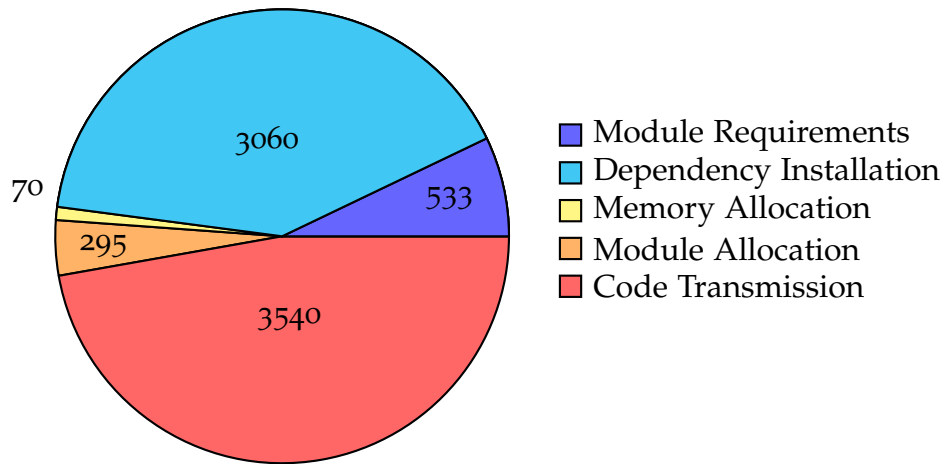


Figure 6.4.: Time portions of modular installation

- **Dependency Installation:** Described in Section 5.4.3.
- **Memory Allocation:** As part of Section 5.4.4, memory is allocated for the module *led\_app*.
- **Module Allocation:** This step includes allocating a module slot as well as writing the Load Information data to flash.
- **Code Transmission:** This test measures how long it takes to transmit the code of *led\_app* to the device, writing it to flash memory and calculating a CRC-8 checksum, as well as sending it back to the Module Server and waiting for confirmation. Given that the amount of data is known to be 527 bytes, the approximate bandwidth can be determined to be 149 bytes per second.

## 7. Summary

This section will summarize the findings and contributions in this thesis by reiterating on the goals of this thesis, defined in Section 1:

- Compile modules for SmartOS on a host machine
- Establish a protocol to transmit modules from a host machine to a device
- Transmit the compiled module in as few bytes as possible
- Load modules on a device during run-time
- Enable module function calls across modules in a dynamic environment

Module Server, which is described in Section 4.2 and implemented in Section 5.1 is capable of cross-compiling modules for the MSP430F5529, based on parameters received within a request from a device. Other architectures are not supported, but given the modular design of it is just a matter of writing a concise wrapper around the compiler itself. But not only different architectures, it is also possible to write modules in languages other than C, as long as a matching backend for the Module Server is implemented. On top of that, the design makes it easy to add new modules and describe their metadata, such as dependencies, versions and supported architectures. Out of the focus of this work was supporting multiple device requests at once. Due to this, only one device can communicate with the Module Server at a time. Larger installations will likely want to address this limitation.

The protocol used for communication between device and Module Server is described in Section 4.3, with more implementation-specific details in Section 5.4. It enables devices to request modules from the Module Server in a reliable way, with transmission of binary data being protected by checksum verification. By sending only the actual code and data sections, data is reduced by about a factor of 7, as shown in Section 6.1.

## 7. Summary

---

Loading modules at run-time required some changes to the SmartOS kernel. First and foremost, there were no facilities to dynamically allocate memory, so Section 5.2 describes the efforts to add this functionality. Following that, SmartOS relied on information about installed modules that was generated during compilation of the kernel. In a dynamic environment, this information is not available, so Section 5.3 describes the implementation of a dynamic module manager that is capable of loading and updating modules at run-time.

Section 4.6 describes the efforts required to loosen the tightly-coupled function calls on SmartOS, based on indirections through addresses stored in arrays. The overhead of this implementation is analyzed in Section 6.2 and uses about four times as many CPU cycles compared to direct function calls.



## 8. Conclusion and Future Work

This thesis improves a design for software updates during run-time on a real-time operating system based on the work in the paper *Towards Automatic SW Integration in Dependable Embedded Systems* [3].

### 8.1. Outlook

There are some points in the previous implementation and design that are not optimal or did not fit the scope of this thesis. This section points out issues and ways to go forward.

#### 8.1.1. Module Server

In the current implementation, the Module Server can only communicate with the currently plugged in device over UART and needs to be restarted before connecting to a different device. This can be improved by monitoring operating system level events on newly plugged in devices and only then opening a connection to them.

UART is only one of many possible communication channels. Depending on the device, implementing wireless methods for updates can be beneficial for rolling out updates to a large number of devices.

As mentioned in Section 4.1.1 this work only focusses on devices of performance class 1. Other classes require implementation of, for instance, the following features:

## 8. Conclusion and Future Work

---

- Tracking of device's memory and module information in the Module Server (DPC-0).
- An efficient relocation algorithm on the device (DPC-2, DPC-3, DPC-4).
- Storing of exported symbols of modules on the device (DPC-3, DPC-4).

As Section 4.2 declares, the current implementation only supports MSP430F5529, whereas SmartOS targets RISC-V and TC297TF as well. Extending the implementation requires creating additional backends for the Module Server on top of changes to the device-side code. The device side of the update protocol is running as a task.

### Dependency Resolution

The currently implemented dependency resolver is very rudimentary and given the very basic versioning scheme is also constrained to performing basic operations. For instance, it is not able to detect cycles in dependency graphs, which will cause it to end up in an endless loop. Other than that, and this is owed to the limited versioning scheme detailed in Section 8.1.3, it is not able to reduce module dependencies based on their compatibility. For instance, two apps could depend on two different versions of the same library. Now, even if the newer version is fully compatible with the older one as it simply fixes a bug, both versions would have to be installed. This wastes space on the device.

#### 8.1.2. Preserving State

One problem this work does not address is preserving state of modules between updates. If a module's internal variables change from one version to the next, for instance an update adds or removes some fields in a struct, it also needs to update this data before being able to continue. In this thesis, it is required that applications restart after an update, starting with a fresh state. In production, this could lead to further problems, such as losing state of sensors or other attached devices. Transforming state from one version to the next could be solved by adding migration functions to each new version. These functions would then be able to access the current state of an

application, apply some logic to transform it to the new format and write that new state to a prepared memory location. Only after this migration finished successfully is the application allowed to continue running.

A similar issue for running tasks arises: when updating a driver, there is the chance that one or more tasks are either just about to jump into or are already inside one of its functions. Would this function now be updated, its address could change, leading to the tasks to jump to an undefined memory region. An approach to solving this could be to signal all tasks that depend on a component about to be updated to enter a safe state. In this safe state, they do not jump into any other module's code, but are allowed to keep running. Once the update is finished and addresses are updated, the tasks are allowed to leave their safe state again.

### 8.1.3. Module IDs & Versioning

As explained in Section 4.2, the Module Spec consists of two 8 bit values. That means, there are 256 unique module identification numbers. For production use, this number could be too low.

Version numbers are quite limited as well. For instance, there could be a new release of a library that fixes a bug in its implementation but has full ABI compatibility otherwise. With the current versioning scheme releasing this this would require increasing the version number by one. Modules depending on this library would then be required to update their list of dependencies and so on. This also hinders further optimization when two modules depend on different, but ABI-compatible versions of a library, as mentioned in Section 8.1.1.

This does not scale well with a large number of modules. This problem is not unique to the embedded domain, it affects other software projects as well. An attempt to solve this is called Semantic Versioning [26], which could also be used in this case. In short, this versioning scheme splits version numbers into three parts, using one part to designate API- and possibly ABI-breaking changes and other parts for bugfix releases.

### 8.1.4. Update Protocol

The update protocol described in Sections 4.3 and 5.4 has some implementation issues detailed in the following.

First of all, dependency installation is not ideal. In the current implementation described in Section 5.4.3 pluggability checking and module installation is intertwined. The device does not ensure that enough memory is available for all dependencies and the module itself before attempting to install them. This means that it would need to remove all modules previously installed if one of the following fails due to insufficient memory.

A better way to solve this could be gathering all space requirements before installation to quickly verify enough space is available. This has one downside: as mentioned in Section 5.4.4, the memory requirements of a module might change after linking, leading to either false-positive or false-negative responses.

Another approach would be to make use of the Module Manager's module status bit `ValidLI`. In normal operation, it is set to 0 at the very end of a successful module installation process. The implementation could be adapted to only flip this bit for each dependency once the final module is installed successfully. In case of an error somewhere in the installation the process will simply abort and the memory can be reused in future installations.

To further reduce the transmitted size of binaries on module updates the Module Server could only transmit differences between versions, which is known as delta update. Especially for large modules or small changes this could be beneficial to transfer times.

### 8.1.5. Memory Manager

As mentioned in Section 4.4 the Memory Manager is a very basic implementation and lacks essential features. The most prominently missing feature is a solution for memory fragmentation. When installing and uninstalling

modules "holes" are left behind in memory, more commonly known as fragmentation. Defragmenting the memory would mean to combine consecutive empty blocks as well as reordering occupied blocks to have all used memory in one place and all free memory following it.



# Appendix





## **Appendix A.**

### **Glossary**



# Glossary

**ABB** Atomic Basic Block. 71

**API** Application Programming Interface. 71

**Application** software that performs a set of tasks to satisfy the requirements of the end user/customer. It is composed of one or more modules. The source code of application modules is hardware-independent, but it can make use of hardware-dependent modules, such as drivers. 71

**COFIE** COntrol Flow and Interaction Expression. 71

**Compatibility Check** pluggability + interoperability checks. 14, 71

**CSP** communicating sequential processes. 71

**Dependencies** OS and other modules (drivers, libraries, etc.) required by a module, including their corresponding versions. 71

**DPC** Device Performance Class. Classification of a device based on its resources and performance. 71

**ELF** Executable and Linking Format. File format for executable binary programs and libraries used on many Linux and UNIX-like operating systems. 71

**Hard Deletion** erasing the content of a portion of memory. 71

**Installation** write an already linked and relocated module into ROM. 71

**Interoperability Check** it is satisfied if an update does not violate any version constraints. 71

**IoT** Internet of Things. The ITU-T <sup>1</sup> defines IoT as

global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual)

---

<sup>1</sup>International Telecommunication Union Telecommunication Standardization Sector

things based on existing and evolving interoperable information and communication technologies. NOTE 1 – Through the exploitation of identification, data capture, processing and communication capabilities, the IoT makes full use of things to offer services to all kinds of applications, whilst ensuring that security and privacy requirements are fulfilled. NOTE 2 – From a broader perspective, the IoT can be perceived as a vision with technological and societal implications [23]. 10, 71

IP intellectual property. 71

LI Load Information. 71

Linking resolution of a module's external symbols. 71

Linux is a generic term referring to the family of Unix-like computer operating systems that use the Linux kernel. 71

Loading initialization of a modules' RAM and OS data structures for execution. 71

Lua Lua is a powerful, efficient, lightweight, embeddable scripting language. It supports procedural programming, object-oriented programming, functional programming, data-driven programming, and data description. [10]. 71

MeM Memory Manager. 14, 71

MI Modules Information. 71

Module piece of software independently developed against the target system's interface, but with no information about its implementation (black box model). A module contains zero or more tasks. For example, within MCSmartOS, drivers and libraries have no tasks; services and applications have one or more tasks. Modules are isolated from each other, but tasks within a module are not. 71

MoM Modules Manager. 71

NFP Non-Functional Property. 71

- OEM Original Equipment Manufacturer. 71
- OS in modular MCSmartOS, the OS is the minimal software to allow a system to run, and to support partial updates. It is composed by the kernel, startup code, interrupt vectors and essential modules (e.g., sysclock and network stack). 71
- PCP priority ceiling protocol. 71
- PIC Position Independent Code. 71
- Pluggability Check it is satisfied if the module to be installed fits into memory and all dependencies are available on the device.. 71
- Relocation resolution of a module's internal symbols. 71
- ROM shorter notation for flash/program memory. 71
- Soft Deletion mark a portion of memory as deleted, without erasing its content. 71
- SWC software component. 71
- Task MCSmartOS' basic schedulable unit, analogous to processes in UNIX systems. Every task has a name, a base priority, a dedicated stack, and an entry point. MCSmartOS builds a Task Control Block (TCB) for each task in the system. 71
- TCB Task Control Block. 71
- WCET Worst Case Execution Time. 71



## Appendix B.

### Code

```

1 def generate_kernel_symbol_map(kernel_path: Path) -> str:
2     output = ""
3     symbols_map = defaultdict(list)
4     with open(kernel_path, "rb") as stream:
5         elf = ELFFile(stream)
6
7         symbols = elf.get_section_by_name(".symtab")
8
9         if not symbols or not isinstance(symbols, SymbolTableSection):
10             raise NoSymbolTableError
11
12         for symbol in symbols.iter_symbols():
13             if (
14                 not symbol.name
15                 or symbol["st_info"]["type"] not in ("STT_FUNC", "STT_OBJECT")
16                 or symbol.name.startswith("__")
17                 or symbol["st_other"]["visibility"] == "STV_HIDDEN"
18             ):
19                 continue
20             section = elf.get_section(symbol["st_shndx"])
21             symbols_map[section.name].append(
22                 (symbol.name, symbol["st_value"], section["sh_addr"])
23             )
24
25         for section_name, symbols in symbols_map.items():
26             tmp_section_name = section_name[1:]
27
28             output += "__{}_start = 0x{:04x};\n".format(tmp_section_name, symbols[0][2])
29             for name, address, section_address in sorted(symbols, key=lambda s: s[1]):
30                 output += "{} = __{}_start + 0x{:04x};\n".format(
31                     name, tmp_section_name, address - section_address
32                 )
33
34         return output

```

Listing B.1.: Extraction of kernel symbols on the Module Server



---

```
1  #include <stdarg.h>
2  #include <ostypes.h>
3  #include <led_driver/led.h>
4
5  extern void smartPrintfWrapper(unsigned char *a, ...);
6  extern void sleepUntil(Time_t deadline);
7  extern void getCurrentTime(Time_t *deadline);
8
9  OS_TASK_CONSTR(LedApp, 450, 20);
10
11  OS_TASKENTRY(LedApp) {
12      Time_t time;
13      led_t leds[] = {LED_01, LED_02};
14      uint8_t led = 0;
15
16      while(1) {
17          led = 1 - led;
18          LEDtoggle(leds[led]);
19          getCurrentTime(&time);
20          uint64_t waitTime = GetBlinkSpeed();
21          time += waitTime;
22          sleepUntil(time);
23          LEDtoggle(leds[led]);
24      }
25  }
```

Listing B.2.: Implementation of led\_app module

```
1  #include "led.h"
2
3  extern Result_t io_configureDirection(uint8_t port, uint8_t pin, uint8_t in_out);
4  extern Result_t io_configureSelection(uint8_t port, uint8_t pin, uint8_t io_peripheral);
5  extern Result_t io_togglePin(uint8_t port, uint8_t pin);
6  extern Result_t io_setPinValue(uint8_t port, uint8_t pin, uint8_t value);
7
8  /// ID: 0
9  Result_t LEDconfigure(led_t led){
10
11     if(io_configureDirection((led >> 4), (led & 0x0F), 1) == FAILURE){
12         return FAILURE;
13     }
14
15     return io_configureSelection((led >> 4), (led & 0x0F), 0u);
16
17 }
18
19 /// ID: 1
20 Result_t LEDtoggle(led_t led){
21     return io_togglePin((led >> 4), (led & 0x0F));
22 }
23
24 /// ID: 2
25 Result_t LEDsetState(led_t led, ledState_t state){
26     return io_setPinValue((led >> 4), (led & 0x0F), state);
27 }
28
29 /// ID: 3
30 uint64_t GetBlinkSpeed() {
31     return 1*500000ul;
32 }
```

Listing B.3.: Implementation of led\_driver module

---

```

1  #include "led_driver/led.h"
2  #include "ostypes.h"
3
4  extern uint8_t* (*TTBL[])(uint8_t);
5
6  Result_t LEDconfigure(led_t a0) {
7      return ((Result_t (*)(led_t))(TTBL[0](0)))(a0);
8  }
9
10 Result_t LEDtoggle(led_t a0) {
11     return ((Result_t (*)(led_t))(TTBL[0](1)))(a0);
12 }
13
14 Result_t LEDsetState(led_t a0, ledState_t a1) {
15     return ((Result_t (*)(led_t, ledState_t))(TTBL[0](2)))(a0, a1);
16 }
17
18 uint64_t GetBlinkSpeed() {
19     return ((uint64_t (*)( ))(TTBL[0](3)))( );
20 }
```

Listing B.4.: Automatically generated stubs of led\_driver for use in led\_app



# Bibliography

- [1] Amazon Web Services, Inc. *FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions*. 2021. URL: <https://www.freertos.org/> (visited on 08/11/2021) (cit. on p. 10).
- [2] Emmanuel Baccelli et al. "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT." In: *IEEE Internet of Things Journal* 5.6 (Dec. 2018), pp. 4428–4440 (cit. on p. 10).
- [3] Leandro Batista Ribeiro, Marcel Carsten Baunach, and Fabian Schlager. "Towards Automatic SW Integration in Dependable Embedded Systems." English. In: *International Conference on Embedded Wireless Systems and Networks (EWSN)*. 17th International Conference on Embedded Wireless Systems and Networks : EWSN 2020, EWSN ; Conference date: 17-02-2020 Through 19-02-2020. Feb. 2020. URL: <https://ewsn2020.conf.citi-lab.fr/> (cit. on pp. 2, 13, 18–21, 44, 51, 63).
- [4] Niels Brouwers, Koen Langendoen, and Peter Corke. "Darjeeling, a Feature-Rich VM for the Resource Poor." In: *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*. SenSys '09. Berkeley, California: Association for Computing Machinery, 2009, pp. 169–182. ISBN: 9781605585192. DOI: [10.1145/1644038.1644056](https://doi.org/10.1145/1644038.1644056). URL: <https://doi.org/10.1145/1644038.1644056> (cit. on p. 24).
- [5] Chris Liechti. *pyserial*. [Online; accessed 2021-09-13]. Nov. 2020 (cit. on p. 16).
- [6] A. Dunkels, B. Gronvall, and T. Voigt. "Contiki - a lightweight and flexible operating system for tiny networked sensors." In: *29th Annual IEEE International Conference on Local Computer Networks*. 2004, pp. 455–462. DOI: [10.1109/LCN.2004.38](https://doi.org/10.1109/LCN.2004.38) (cit. on p. 9).

- [7] Eli Bendersky. *pyelftools*. [Online; accessed 2021-09-13]. Oct. 2020 (cit. on p. 16).
- [8] Stuart Feldman. *make*. 1976 (cit. on p. 13).
- [9] Damien George. *MicroPython - Python for microcontrollers*. 2018. URL: <https://micropython.org/> (visited on 08/11/2021) (cit. on p. 24).
- [10] Roberto Ierusalimschy, Waldemar Celes, and Luiz Henrique de Figueiredo. *The Programming Language Lua*. 2021. URL: <https://www.lua.org/about.html> (visited on 08/11/2021) (cit. on pp. 10, 24, 74).
- [11] Texas Instruments. *MSP430x5xx and MSP430x6xx Family - User's Guide*. 2018. URL: <http://www.ti.com/lit/ug/slau208q/slau208q.pdf> (cit. on pp. 18, 35, 55, 57).
- [12] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation." In: San Jose, CA, USA, Mar. 2004, pp. 75–88 (cit. on p. 16).
- [13] Philip Levis and David Culler. "Maté: A Tiny Virtual Machine for Sensor Networks." In: *SIGOPS Oper. Syst. Rev.* 36.5 (Oct. 2002), pp. 85–95. ISSN: 0163-5980. DOI: 10.1145/635508.605407. URL: <https://doi.org/10.1145/635508.605407> (cit. on p. 24).
- [14] *Loose Coupling and Architectural Implications*. Institute of Architecture of Application Systems, University of Stuttgart. Sept. 2016 (cit. on p. 1).
- [15] LLVM team - pypi upload by Loic Jaquemet. *clang*. [Online; accessed 2021-09-13]. Aug. 2020 (cit. on p. 16).
- [16] Bogdan Marinescu and Dado Sutter. *Embedded power, driven by Lua*. 2011. URL: <http://www.eluaproject.net/> (visited on 08/11/2021) (cit. on p. 24).
- [17] Renata Martins Gomes, Marcel Carsten Baunach, and Leandro Batista Ribeiro. "MCSSmartOS: A Dependable OS for Compositional Embedded Systems." English. In: FoE-Tag des Field of Expertise "Information, Communication and Computing" ; Conference date: 28-03-2017. Mar. 2017 (cit. on p. 1).
- [18] Zephyr project members. *Zephyr RTOS*. 2022. URL: <http://www.zephyrproject.org/> (visited on 09/03/2022) (cit. on p. 10).

- [19] *MSP430 Embedded Application Binary Interface*. Texas Instruments. June 2013. URL: <https://www.ti.com/lit/an/slaa534a/slaa534a.pdf> (cit. on p. 52).
- [20] *MSP430F552x, MSP430F551x - Mixed-Signal Microcontrollers*. Texas Instruments. Sept. 2018. URL: <http://www.ti.com/lit/ds/symlink/msp430f5529.pdf> (cit. on pp. 16, 19).
- [21] W. Munawar et al. "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](https://doi.org/10.1109/ICC.2010.5501964) (cit. on p. 17).
- [22] George Oikonomou et al. "The Contiki-NG open source operating system for next generation IoT devices." In: *SoftwareX* 18 (2022), p. 101089. ISSN: 2352-7110. DOI: [10.1016/j.softx.2022.101089](https://doi.org/10.1016/j.softx.2022.101089) (cit. on p. 10).
- [23] *Overview of the Internet of things*. Tech. rep. June 2012. URL: <http://handle.itu.int/11.1002/1000/11559> (cit. on p. 74).
- [24] Peter Ruckebusch et al. "GITAR: Generic extension for Internet-of-Things ARchitectures enabling dynamic updates of network and application modules." In: *Ad Hoc Networks* (June 10, 2015) (cit. on p. 23).
- [25] Pipenv maintainer team. *pipenv*. [Online; accessed 2021-09-13]. Nov. 2020 (cit. on p. 16).
- [26] Tom Preston-Werner. "Semantic Versioning 2.0.0." In: *línea*. Available: <http://semver.org> (2013) (cit. on p. 65).