HAW HAMBURG

BACHELOR THESIS

Lasse Jonas Rosenow

Runtime configuration of constrained devices via a shared operating system module

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Lasse Jonas Rosenow

Runtime configuration of constrained devices via a shared operating system module

Bachelor thesis submitted for examination in Bachelor´s degree
in the study course *Bachelor of Science Informatik Technischer Systeme*
at the Department Computer Science
at the Faculty of Engineering and Computer Science
at University of Applied Science Hamburg

Supervisor: Prof. Dr. Thomas Schmidt
Supervisor: Prof. Dr. Franz Korf

Submitted on: February 1, 2023

Lasse Jonas Rosenow

Title of thesis

Runtime configuration of constrained devices via a shared operating system module

Keywords

Runtime Configuration, Constrained Devices, IoT, Operating System

Abstract

Many applications on the Internet of Things (IoT) use parameters that need to be changed at runtime. Typical examples can be authentication credentials, the sampling rate of a measurement or an LED color. Furthermore in modern industrial spaces runtime parameters of devices are remotely changeable using plug-and-play capable protocols such as LwM2M etc.

As of today, RIOT OS does not provide an API for runtime parameters. Instead, each application needs to implement custom, often redundant logic for its specific use case. A fully featured integration of external Configuration Managers such as LwM2M proves to be particularly difficult given the vast amount of drivers and devices that need to be supported and maintained for every single management tool.

In this thesis we introduce a system-level configuration registry that allows for setting and getting configuration parameter values at runtime. It organizes parameters according to a standardized schema that includes type information and other metadata such as name and description fields to make them accessible to other modules such as Configuration Managers. The parameter values are optionally persisted on local storage. To change the parameters we specify a CLI and interfaces to CoAP, MQTT and LwM2M. Furthermore, we discuss important API design decisions and explain why standardized, module-independent schemas for common configuration parameters are essential for the integration of external management tools such as LwM2M.

Lasse Jonas Rosenow

Thema der Arbeit

Laufzeitkonfiguration von eingeschränkten Geräten über ein gemeinsames Betriebssystemmodul

Stichworte

Laufzeitkonfiguration, Eingebettete Geräte, IoT, Betriebssystem

Kurzzusammenfassung

Viele Anwendungen im Internet der Dinge (IoT) verwenden Parameter, die zur Laufzeit geändert werden müssen. Typische Beispiele können Authentifizierungsdaten, die Abtastrate einer Messung oder eine LED-Farbe sein. Darüber hinaus sind in modernen Industrieumgebungen Laufzeitparameter von Geräten über Plug-and-Play-fähige Protokolle wie LwM2M usw. fernveränderbar.

Gegenwärtig stellt RIOT OS keine API für Laufzeitparameter bereit. Stattdessen muss jede Anwendung benutzerdefinierte, oft redundante Logik für ihren spezifischen Anwendungsfall implementieren. Eine voll funktionsfähige Integration externer Konfigurationsmanager wie LwM2M erweist sich angesichts der enormen Menge an Treibern und Geräten, die für jedes einzelne Verwaltungstool unterstützt und gewartet werden müssen, als besonders schwierig.

In dieser Arbeit wird eine Konfigurationsregistrierung auf Systemebene eingeführt, mit der man Parameterwerte zur Laufzeit setzen und abzurufen kann. Sie organisiert die Parameter nach einem standardisierten Schema, das Typinformationen und andere Metadaten wie Name und Beschreibungsfelder enthält, um sie anderen Modulen wie Konfigurationsmanagern zugänglich zu machen. Die Parameterwerte werden optional im lokalen Speicher gehalten. Um die Parameter zu ändern, werden eine CLI und Schnittstellen zu CoAP, MQTT und LwM2M spezifiziert. Darüber hinaus werden in dieser Arbeit wichtige Entscheidungen zum API-Design diskutiert und erläutert, warum standardisierte, modulunabhängige Schemata für gemeinsame Konfigurationsparameter für die Integration von externen Management-Tools wie LwM2M notwendig sind.

Contents

List of Figures

List of Tables

Listings

Abbreviations

API	Application Programming Interface.
CBOR	Concise Binary Object Representation [2].
CG	Configuration Group.
CLI	Command-Line Interface.
CN	Configuration Namespace.
CoAP	Constrained Application Protocol [1].
CP	Configuration Path.
CS	Configuration Schema.
IoT	Internet of Things.
JSON	JavaScript Object Notation [3].
Mynewt Config	Apache Mynewt Config subsystem [4].
RCS	Runtime Configuration System.
SF	Storage Facility.
SI	Schema Instance.
UART	Universal Asynchronous Receiver Transmitter.
VFS	Virtual File System.

WLAN	Wireless Local Area Network.
YANG	Yet Another Next Generation.

1 Introduction

1.1 Context

For constrained devices, it can be important not only to set certain configurations before compiling and flashing a device but also while the device is in full operation, for example to change the rate or precision of a measurement or to update authentication credentials. Also, it is often required to persist these values, so that a device would keep its configuration after a restart caused by a firmware update or power shortage for example. It is also a common requirement to change these runtime configuration parameters via an external Configuration Manager that communicates with the constrained device through WLAN, Bluetooth, a serial bus, etc. Not only smart home, but also more and more smart city applications require this external configuration management to be based on a standardized protocol such as LwM2M [5], or the proprietary "matter" standard [6] in smart home applications. Using a standardized protocol gives the advantage of being able to automatically integrate a node into already existing systems without the need to write custom wrappers and also having the freedom to replace them with compatible devices implementing the same protocol.

1.2 The Problem of Runtime Configuration in RIOT OS

As of today, there is no official way within RIOT OS [7] for modules to expose runtime configurable parameters, as can be seen in its official documentation [8]. As a consequence, every application needs to implement its own way to work around this, causing a lot of duplicated work for all vendors using RIOT OS and also leading to large inconsistencies in their custom implementations making it impossible to easily integrate different devices. For example, if there is a smart bulb vendor "A" and a smart bulb vendor "B", which both need to be controlled by a control unit made by vendor "C", this control unit would need to implement the custom API of vendor “A” and vendor “B”. Causing not only a huge amount of unnecessary work for vendor “A” and vendor “B” but also vendor “C”.

1.3 Thesis Aims and Objectives

In this thesis we aim to investigate how to design an architecture that will extend the RIOT OS so that modules can expose configuration parameters in a way that they can easily integrate with external management tools such as LwM2M. We also want to persist these values beyond device restarts.

1.4 Thesis Structure

In chapter 2 we start by explaining the importance of configuration for constrained devices and the differences between static and runtime configuration. Furthermore, we discuss why it is beneficial to implement this feature on an operating system level instead of per application.

In chapter 3 we explain what kind of requirements the new RIOT OS Registry is supposed to fulfill and discuss why these requirements are needed.

In chapter 4 we first present existing academic work and discuss similarities and differences to our requirements. Then we present existing implementation work and analyze in how much each of it fulfills the in chapter 3 explained requirements. After that, we conclude if one of the previously assessed work already fulfills enough requirements to just be implemented inside RIOT OS as its official Runtime Configuration System (RCS), or if the architecture for the RIOT OS Registry will only be based on one of the previously analyzed work but extend its functionality, or if a completely new architecture needs to be designed and implemented.

In chapter 5 we present the design of the RIOT OS Registry that fulfills all the requirements from chapter 3. We explain all the behavioral aspects, API decisions, and how the specific requirements from chapter 3 are fulfilled.

In chapter 6 we explain the implementation of the in chapter 5 presented architecture, going more into detail as to how and why the more technical decisions of the API design were made.

In chapter 7 we explain how the correctness of the main API functions of the in chapter 6 implemented source code is tested.

In chapter 8 we evaluate the overhead of the RIOT OS Registry implementation (see chapter 6). First, we measure the stack consumption of the main RIOT OS Registry functions, then we discuss the results of these measurements. Second, we measure the ROM size of the RIOT OS Registry implementation and also discuss our findings.

In chapter 9 we explain which steps are next to continue on the work of this thesis and how to further iterate upon it.

In chapter 10 we conclude on how successful this work is in fulfilling its aims set in section 1.3 and more specifically in chapter 3 and reflect on what lessons we have learned in the process.

2 Background of (OS) Configuration

2.1 The Importance of Configuration

Configuration is a way of making a program flexible, so it can be used in scenarios that have different requirements and are not compatible with each other, without the need to maintain multiple versions of the program that are almost identical. It i.e. allows hiding certain features behind a configuration flag or gives the possibility to change some internal values by exposing them via an external interface.

2.2 Static vs. Dynamic Configuration

Most applications use static configuration parameters to become more flexible. These static configuration parameters are set before the source code gets compiled to machine code and can not be changed after. They are constant values written to the program storage. Typical use-cases are flags to enable/disable program features or parameters to configure which pins to use for i.e. I^2C on a hardware board. In the RIOT OS, this is typically done with the help of CFLAGS inside a Makefile [9] or through Kconfig [10] on more modern setups. The benefit of static configuration is that in cases in which a dynamic configuration has no benefit, these configuration values are less simple to accidentally being mutated by some parts of the application as they can not be changed. In general, every configuration parameter that does not need to be changed at runtime, as seen in the examples above, should be defined statically.

For some configuration parameters though it is necessary to be able to change them at runtime. These parameters must be dynamic and not static. A common use-case is a “smart” RGB bulb. If its color value is static, it can only be changed by recompiling the program of the RGB bulb and flashing it to the RGB bulb flash storage. But if it is dynamic, it can be changed at runtime, without the need to recompile the whole application. To change runtime configuration parameters, they can be for example exposed through a CoAP [1] or LwM2M interface to enable external control.

Static and dynamic configurations can also be mixed by dynamically switching between a set of statically defined configurations. For example, an application could run in “IDLE” or “PERF” mode. Then it can be statically defined, what CPU frequency etc. must be used for “IDLE” or “PERF”. It is possible to dynamically switch between these predefined modes [11].

2.3 Benefits of an Operating System-Level Implementation

Implementing a runtime configuration registry on an operating system level means that it can also benefit from the hardware abstractions provided by it. For example, if the RIOT OS Registry implements a CoAP API to enable external access to configuration values, this CoAP API relies on the RIOT OS network stack [12]. This network stack is then implemented by all the different target devices supported by RIOT OS. This way the registry works on all the devices supported by RIOT OS and does not need to implement wrappers for every target device itself.

Besides the hardware abstraction, an operating system level implementation also comes with the benefit that it can integrate with all the modules/drivers that are already provided by the operating system itself [13]. This way a RIOT OS application can just enable the registry feature and automatically get runtime configuration capabilities for all drivers or modules it depends on. If the registry is written on an application level, it needs to implement the drivers/modules itself and then integrate them with the registry on a per-application basis, instead of writing it once for the operating system.

3 Requirements of a Runtime Configuration System (RCS)

To be able to assess how suitable existing related work is, or if it is necessary to design a completely new architecture to fulfill the requirements for runtime configuration inside RIOT OS, it is first necessary to define what is needed and why.

3.1 Shared Configuration Schemas

A Configuration Schema (CS) is mostly a data structure combined with some additional metadata such as parameter types or preconditions. For a Configuration Manager to be easily integrated into other external tools, it is necessary to have a collection of CSs to describe common configuration needs. For example, there could be an LED CS consisting of three unsigned 8-bit integer variables, or a WI-FI CS containing the SSID, the password, etc. Now, these CSs are supposed to be implemented by for example LED drivers, or WI-FI module drivers. This way the registry has a consistent API, as can be seen on the right-hand side of Figure 3.1.

An alternative approach is to define a custom CS per module basis, as can be seen on the left-hand side of Figure 3.1. This results in a lot of duplicated work and inconsistent APIs for equal operations. The advantage of this approach is that the work put into defining the custom CS is less than specifying a CS that has to fit for all comparable use-cases. So new modules/driver that require a certain configuration module that is for example not yet implemented in the registry would just specify their own and don't need to find a solution for all first. This would save a lot of review time and helps adoption.

If each approach's advantages and disadvantages are compared, having a consistent API outweighs the time saving of a per-module approach and is worth spending some extra time for, as this extra work is not wasted, but essential to enable the integration of external Configuration Managers. With no consistent API, those tools would need to be adapted to every single driver instead of just the CSs, that are shared among modules.

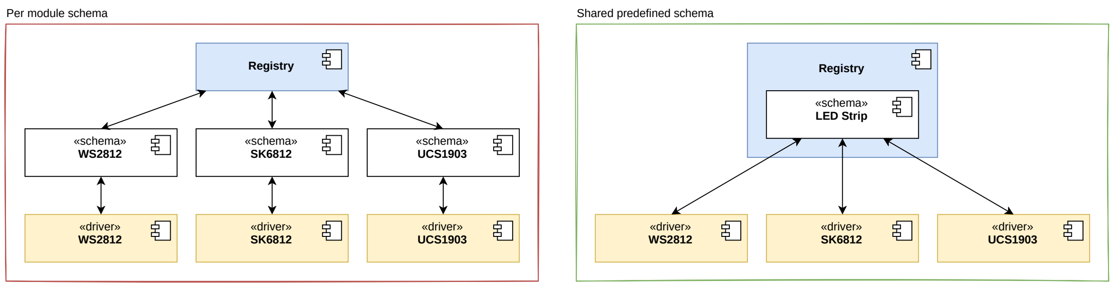


Figure 3.1: Per module CSs (left-hand side) and Shared CSs (right-hand side).

3.2 Multiple Instances per Configuration Schema

This requirement is strongly connected to section 3.1. Having shared CSs requires that these can be used at the same time, by multiple modules, as can be seen on the right-hand side of Figure 3.2, without causing conflicts as can be seen on the left-hand side of Figure 3.2. If an example application has 3 different LED modules, of which each implements the same LED CS, they need to expose their data in their instance of this CS and don't overwrite each other's configurations. Besides that even if there is only one module, that is used to control three LEDs of the same kind, it is also necessary for this module to create multiple instances so that each LED can be configured on its own.

This means that the CS should not hold the actual data but only specify its structure. Modules then shall be able to create as many instances of this CS as is needed.

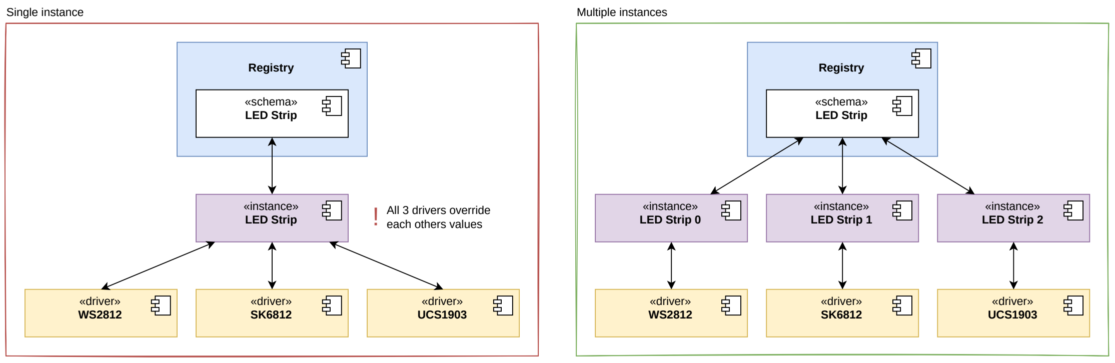


Figure 3.2: Single shared instance (left-hand side) and Multiple instances (right-hand side).

3.3 Integer Path as the Identifier of Configuration Values

To uniquely identify each configuration parameter a path or array is needed that points to the parameter. For example “schema_id/instance_id/parameter_id”.

One way to do this is using an array of strings, as can be seen on the left-hand side of Figure 3.3, inspired by the URI in HTTP [14]. This approach is very verbose, which is good for integrations such as MQTT [15] or CoAP, which also rely on using string based identifier paths. It is also significantly less difficult for humans to work with compared to numbers for example. The downside of using strings is a huge amount of overhead, especially for constrained devices. Not only in terms of processing power that is necessary to path strings compared to simple arrays of numbers, but most importantly in terms of unnecessary payload when accessed remotely. For example through LoRA, which in some scenarios could make every single byte count [16].

Using integers instead of strings does not come with these issues, but also has some disadvantages, such as a lower human readability, as can be seen on the right-hand side of Figure 3.3. But RIOT OS is an operating system for constrained devices and this is why low connectivity and low power scenarios are more important use-cases than human readability. Also, if the parameter identifier is based on an integer path, it is possible to give parameters optional string identifiers to improve the integration of external tools and also improve human readability. (Besides that modern IoT configuration protocols such as the CoAP Management Interface (CORECONF) [17] also use integers as identifiers, which does not mean it is the right to choose in this case, but still strengthens the point.)

As a conclusion, integer arrays with optional string metadata fields are a solid solution for runtime configuration parameter identifiers of constrained devices.

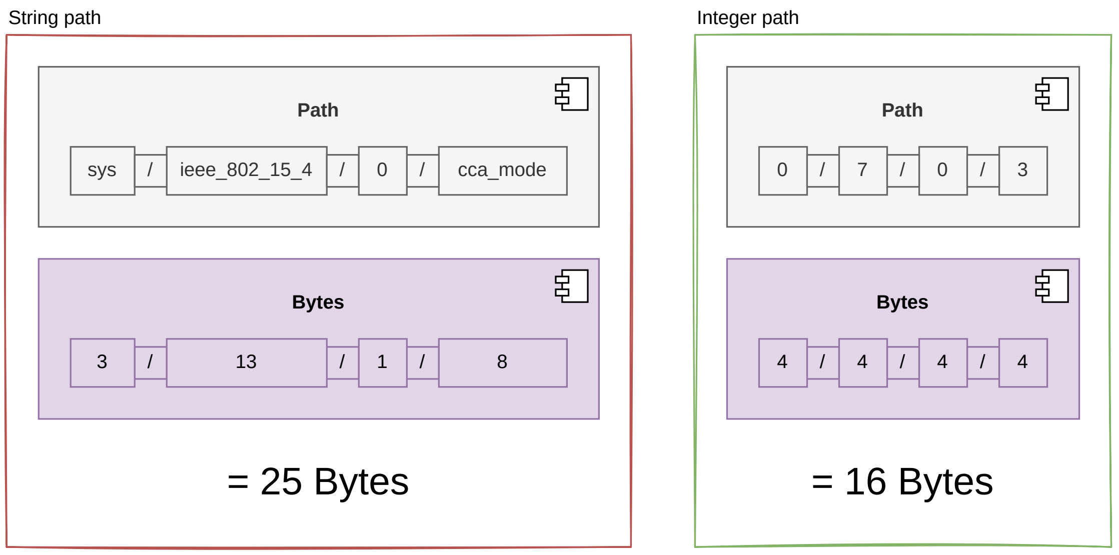


Figure 3.3: String path (left-hand side) and Integer path (right-hand side).

3.4 Nested Configuration Groups

A CS can either have a nested file system like structure of Configuration Groups (CGs) (folders) that contain parameters (files) or even more CGs as can be seen on the right-hand side of Figure 3.4, or on the other hand, it can also be implemented just as a flat key-value structure as can be seen on the left-hand side of Figure 3.4.

From the implementation perspective, a key-value structure is less difficult to implement, but it gives the CS less flexibility and could cause workarounds such as long parameter name tags. For example “group1_group2_group3_parameter0” or “group2_group9_-group7_parameter5”.

As a conclusion, to give more flexibility it is preferred to have the ability to specify nested configuration structures, but it is not an important requirement and does not decide whether a possible implementation is suitable or not.

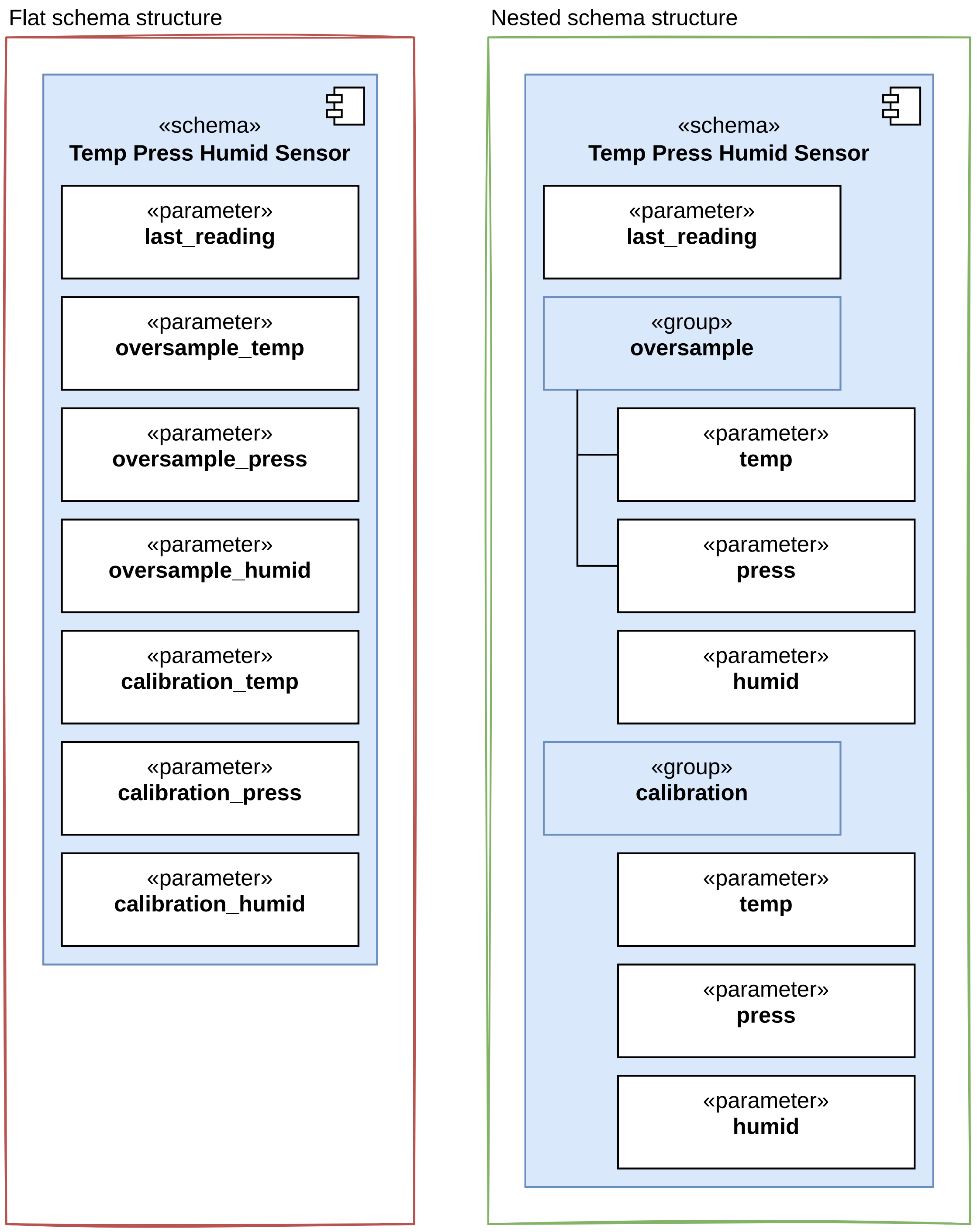


Figure 3.4: Flat schema structure (left-hand side) and Nested schema structure (right-hand side)

3.5 Typed Configuration Parameters

The types of configuration parameters must be exposed as can be seen on the right-hand side of Figure 3.5. This allows defining typed external APIs.

If the type of a configuration parameter is not exposed as can be seen in Figure 3.5, it can not be passed on to external APIs, allowing errors to occur caused by incompatible input data.

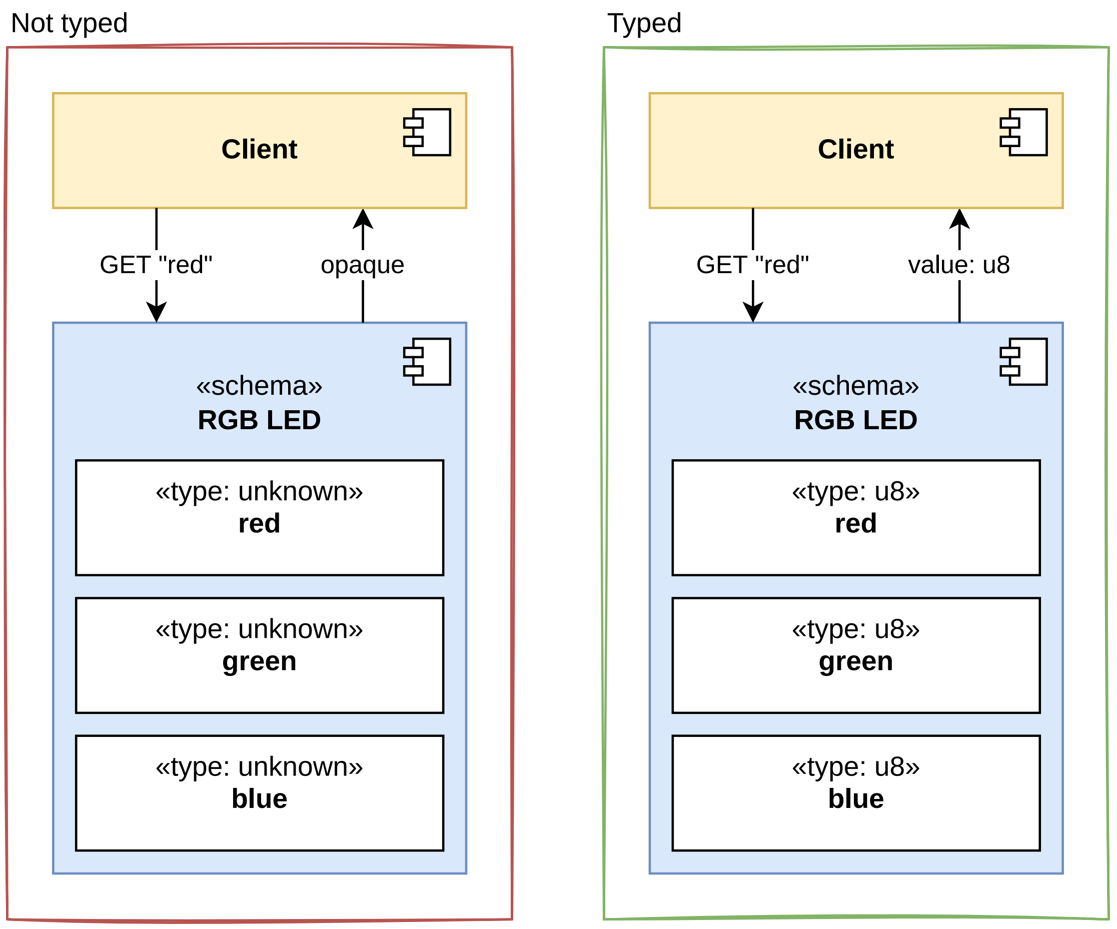


Figure 3.5: Not typed parameters (left-hand side) and Typed parameters (right-hand side).

3.6 Binary Internal Configuration Parameter Value Format

Internally the values of the configuration parameters should be stored and passed around in their binary representation (their correct c type) as can be seen on the right-hand side of Figure 3.6 and not converted to some inefficient format such as a string as can be seen on the left-hand side of Figure 3.6.

The reason for this is that strings have the following drawbacks that are especially problematic with constrained devices:

1. In most cases strings consume significantly more storage than the corresponding types needed to represent the same data.
2. Additionally, heap allocation should be avoided on constrained devices as their storage is usually small and if a program can run or not is best to be known by making sure if the binary fits on the storage or not. So to work with string values internally implies always storing the maximum allowed string length for each parameter, to avoid dynamic heap allocation. This causes an unnecessarily high amount of storage and possibly stack overhead. By using the concrete types, the size of each parameter is always exactly known and is never too large.
3. Converting from- and to a string, or comparing strings causes a lot of computing overhead that can be avoided.

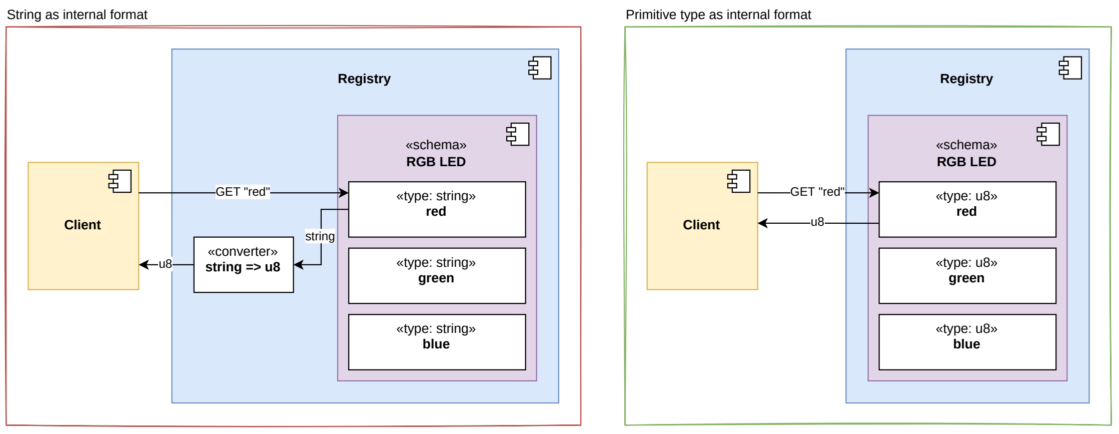


Figure 3.6: “String as an internal format” (left-hand side) and Primitive type as an internal format (right-hand side).

3.7 Transactionally Commit Configuration Changes

In some situations it is necessary that multiple configuration parameters change their value at the same time. For example these configuration parameters depend on each other, such as an RGB LED that has three configuration parameters, one for red, one for green and one for blue. If these parameters don't all change at the same time, the color of the RGB LED will not go from for example red to blue, but from red to black and then to blue or from red to pink and then to blue. So the RCS must have the ability to fulfill this need by committing multiple configuration parameter changes in a transactional way.

3.8 Persistent Configurations

Once a configuration parameter's value is changed, it muste be possible to persist this change on a non-volatile storage. This is important because especially constrained Internet of Things (IoT) devices can have power losses, for example if they are operated by solar energy.

3.9 Low Implementation Effort for Modules/Drivers

The implementation effort that is necessary to get the RCS integrated into each module must be as low as possible while still fulfilling all requirements defined in this section.

If the implementation cost becomes too high, this not only makes maintainability of modules/drivers harder but lowers the chance of high adoption of this runtime configuration module into other drivers/modules in the first place.

3.10 Integration with External Configuration Managers

The RCS must be compatible with common external Configuration Managers. The runtime configuration module is not supposed to specify how external configuration management works for example by introducing an official CoAP API but is thought of as an internal API that can be used by external Configuration Manager modules that then expose this API however needed.

Currently planned external Configuration Manager integrations, to which this internal runtime configuration module must be compatible to are:

- A LwM2M server [5]
- A custom CoAP based API [1]
- A custom MQTT based API [15]

4 Related Work

4.1 Academic Work

4.1.1 Model-driven Development of Adaptive IoT Systems

The paper with the title “Model-driven Development of Adaptive IoT Systems”[18] shows how to develop adaptive IoT systems using a model-driven approach. Its approach is to first model state machines of all system components using SysML4IoT [19], then use a publish/subscribe architecture to model the environment information and the relationship within the system. From these state machines then the source code is generated that can be deployed to the numerous IoT devices. Those devices then change their configuration at runtime (switching to another state), if the system sends them a message that contains information that leads to a state change based on the state machine implementation. Also the system state is synchronized with the previously designed model. This way the system state can be manually changed by changing the state of the model.

Discussion

The approach how to handle runtime configurations of this paper is very different to our approach. This papers designs the system as a whole first, having the relations and adaptions of the nodes in this system in mind and also providing manual runtime configuration through its model. This creates a system that can work very well on its own, but is not very good in “integrating with the world”. Our goal in this thesis is to create a RCS for an operating system (RIOT OS), that is constructed the other way around. Not having the final system in mind, but providing standardized APIs to allow writing mappings to many standardized external Configuration Managers or protocols like LwM2M.

4.1.2 Architecting Emergent Configurations in the Internet of Things

The paper with the title “Architecting Emergent Configurations in the Internet of Things”[20] explains the Emergent Configurations concept and proposes an architecture for its realization. It further focuses on how Emergent Configurations are formed to achieve user’s goals and how applications can adapt to runtime context changes. The main idea behind Emergent Configurations is that all devices integrate with each other and change their configuration / behavior depending on other devices. The paper gives a conference room as an example, which automatically adjusts its curtains if the projector is used and depending on what kind of media is presented and if it is properly visible or not.

Discussion

The connection between this work and the RCS for RIOT OS is, that it shows what can be achieved when having a well designed RCS, that uses shared CSs (see section 3.1)). Without these stable and reliable interfaces that a shared CS offers, it would be impossible for so many different devices to communicate so flawlessly as in this paper.

4.1.3 CoAP Management Interface (CORECONF)

The paper with the title “CoAP Management Interface (CORECONF)” [17] specifies a “network management interface for constrained devices and networks”. It builds on CoAP to access resources specified in a Concise Binary Object Representation [2] (CBOR) mapping of YANG schemas [21] and also converts the YANG identifier string to numeric identifiers to save payload size. If specified, YANG schemas can have multiple instances.

Discussion

This work is similar to the requirements specified in chapter 3 of our thesis, in that it uses integers as a configuration resource identifier and uses shared configuration schemas (YANG), which support multiple instances. The paper differs to the requirements of our thesis in that it does not specify how these configuration changes should be applied / handled by the local (constrained) device itself. The paper only specifies a protocol for how to do configurations through the network. The introduced CORECONF protocol can be used for external configuration management of the new RIOT OS RCS.

4.2 Implementation Work

4.2.1 Apache Mynewt: Config

Mynewt is an embedded OS developed by the Apache Software Foundation. It is similar to RIOT OS and already comes with a system module for runtime configurations, called "Config" [4].

It manages configuration parameters as key-value pairs of strings. A key contains a path to a configuration parameter inside a module. For example the key "id/serial" means the "serial" parameter of the "id" module. So every module that uses Apache Mynewt Config subsystem [4] (Mynewt Config), has its own namespace (initial module name, in this case "id") to put configuration data and how the configuration path is structured after the namespace is defined by each module itself. For example the "id" module could have a configuration parameter under the "id/serial/i2c/instance_1" path. It is also possible to optionally persist configuration values to storage, so that configurations are not lost even after a restart of a device.

Internal: Handler

Each module needs to implement a so called "handler" so it can expose configuration parameters to Mynewt Config. These handlers have a simple API:

"ch_get" Function

This function takes a string key as its input and returns the configuration parameter's value as a string.

"ch_set" Function

This function takes a string key and a string value as its input and sets the configuration parameters value to the given value.

"ch_commit" Function

This function takes a string key as its input, which does not need to point to a concrete configuration parameter, but could also only point to the module itself or some shorter path, because this function will be executed on every configuration parameter, that is within this path. For example the path "id/serial" includes "id/serial/i2c" and also "id/serial/spi". This function takes changes that have been previously made by calling "ch_set" into effect. Just calling "ch_set" only sets a value, but does not apply it. This way multiple configuration parameter can be set to new values one by one and then are taken into effect at the same time.

"ch_export" Function

This function takes a string key and a callback function as its input, which does not need to point to a concrete configuration parameter, but could also only point to the module itself or some shorter path, because this function will be executed on every configuration parameter, that is within this path. For example the path "id/serial" includes "id/serial/i2c" and also "id/serial/spi". This function finds all configuration parameters, that are within the given path and exports them by calling the given callback function and passing their path and their value as an argument.

API

The API contains of 6 basic functions, that cover the most important use cases. It also includes a few more functions for example to help converting configuration values to and from strings, but these are not important for our thesis.

Get Configuration Values

It is possible to get the value of a configuration parameter by calling the "conf_get_-value" function. Internally this function finds the "handler" by the first element of the given path and calls its "ch_get" function.

Set Configuration Values

It is possible to set a value of a configuration parameter to a new value by calling the “conf_set_value” function. Internally this function finds the “handler” by the first element of the given path and calls its “ch_set” function.

Transactionally Commit Configuration Values

It is possible to commit configuration parameters by calling the “conf_commit” function. Internally this function finds the “handler” by the first element of the given path and calls its “ch_commit” function.

Save Configuration Values to Storage

It is possible to save configuration parameters to storage by calling the “conf_save” function. Internally this function finds the “handler” by the first element of the given path and calls its “ch_export” function, by giving a internal function as its callback value, that then gets called by the handler’s export function and writes all the configuration parameters to storage.

Load Configuration Values from Storage

It is possible to load configuration parameters from storage by calling the “conf_load” or “conf_load_one” function. Internally these functions look into the storage and search for either, in case of “conf_load_one”, for a specific configuration parameter, or in case of “conf_load”, for all available configuration parameters. For each parameter, that is found in storage, the “conf_set_value” function is then called.

4.2.2 Zephyr: Settings

Zephyr is one of the most popular current IoT operating systems. It is part of the Linux Foundation and backed by large companies such as Google, Meta, Intel and others [22]. Given its huge success, it is interesting to see how this rather large competitor handles runtime configuration. The system module responsible for runtime configuration within the Zephyr operating system is called Zephyr Settings [23] and an implementation of Mynewt Config [4]. Both APIs are mostly the same but have some minor differences.

Zephyr Settings vs. Mynewt Config - API Comparison:

Get Configuration Values

Table 4.1 shows the similarities between the "get" functions of Mynewt Config and Zephyr Settings. The only real differences besides the usage of different names being that Zephyr settings returns an integer, while Mynewt Config returns a char pointer and that the Zephyr settings API does not limit itself to passing only strings, but accepts a void pointer and has an additional "len" parameter, containing the size of the value.

M. Config	char *conf_get_value(char *name, char *buf, int buf_len);
Z. Settings	int settings_runtime_get(const char *name, void *data, size_t len);

Table 4.1: M. Config vs. Z. Settings: Read

Set Configuration Values

Table 4.2 shows the similarities between the "set" functions of Mynewt Config and Zephyr Settings. The only real differences besides the usage of different names being that Zephyr settings API does not limit itself to passing only strings, but accepts a void pointer and has an additional "len" parameter, containing the size of the new value.

M. Config	int conf_set_value(char *name, char *val_str);
Z. Settings	int settings_runtime_set(const char *name, const void *data, size_t *len);

Table 4.2: M. Config vs. Z. Settings: Write

Transactionally Commit Configuration Values

Table 4.3 shows the similarities between the "commit" functions of Mynewt Config and Zephyr Settings. The only real differences besides the usage of different names being that Zephyr settings API uses a const parameter.

M. Config	int conf_commit(char *name);
Z. Settings	int settings_runtime_commit(const char *name);

Table 4.3: M. Config vs. Z. Settings: Apply

Load Configuration Values from Storage

Table 4.4 shows the similarities between the “load” functions of Mynewt Config and Zephyr Settings. The only real differences is the usage of different names.

M. Config	int conf_load(void);
Z. Settings	int settings_load(void);

Table 4.4: M. Config vs. Z. Settings: Load

Table 4.5 shows the similarities between the “load_one” functions of Mynewt Config and Zephyr Settings. The only real differences besides the usage of different names being that Zephyr settings API uses a const parameter.

M. Config	int conf_load_one(char *name);
Z. Settings	int settings_load_subtree(const char *subtree);

Table 4.5: M. Config vs. Z. Settings: Load a single parameter

Save Configuration Values to Storage

Table 4.6 shows the similarities between the “save” functions of Mynewt Config and Zephyr Settings. The only real differences is the usage of different names.

M. Config	int int conf_save(void);
Z. Settings	int settings_save(void);

Table 4.6: M. Config vs. Z. Settings: Save

Table 4.7 shows the similarities between the “save_one” functions of Mynewt Config and Zephyr Settings. The only real differences besides the usage of different names being that Zephyr settings API does not limit itself to passing only strings, but accepts a void pointer and has an additional “len” parameter, containing the size of the new value.

M. Config	int conf_save_one(const char *name, char *var);
Z. Settings	int settings_save_one(const char *name, const void *value, size_t val_len);

Table 4.7: M. Config vs. Z. Settings: Save a single parameter

4.2.3 LwM2M Object and Resource Registry

While LwM2M by itself is a protocol for external runtime configuration management and not an operating system registry, it still has some appealing aspects that are relevant to the RIOT OS Registry. Besides the fact that the RIOT OS Registry is supposed to integrate external Configuration Managers such as LwM2M. Furthermore, it is to be evaluated if an LwM2M client such as Eclipse Wakaama [24] can be used as the official RIOT OS configuration registry. This way LwM2M is supposed to become the standard for RIOT OS configuration management and other management systems count integrate with LwM2M instead of a RIOT OS-specific registry.

Predefined and typed object definitions

LwM2M comes with its own predefined object definitions. Each object definition has its own unique ID, name, and other metadata and contains multiple properties (configuration parameters) that describe the object and can be read (or written) to. The properties themselves can not contain further child properties. So the property list of an object definition is always flat. Each property has an ID, a name, a type such as Integer, Boolean, String etc. and other metadata. For example, there is the object with the ID 3420. Its name is “LED color light” and it has one property “RGB value” with the ID 1. This property has the type String and expects a color in the RGB hex format (#rrggbb).

Multiple instances

The LwM2M protocol allows an object definition to have multiple instances. So multiple modules/drivers can implement the same LwM2M object or expose multiple instances of themselves that will be accessible at the same time. For example, a smart lamp might have multiple LEDs that all expose the same interface, but should be addressed individually to enable the mixing of different colors.

Integer Resource Path

To identify a property of some object LwM2M uses a path of 3 integers. First comes the object ID, which is the ID of the object definition. Second comes the instance ID which is the ID of the very instance of that object, since there can be multiple instances of each object. And last comes the property ID.

4.2.4 Prior Work on RIOT OS

Before the work on this thesis started, there has already been an initial implementation of Mynewt Config (see subsection 4.2.1) for RIOT OS as an open PR on its GitHub repository (see PR 10622 [25] and PR 10799 [26]).

4.3 Assessment of Implementation Work on Thesis's RCS Requirements

Besides RIOT OS there are other operating Systems such as Zephyr [22], that may already provide solutions for managing runtime configurations in IoT. So to find a suitable architecture to manage runtime configurations in RIOT OS, it is important to evaluate, if related work already exists, that can fully satisfy all the needs listed in chapter 3. Besides that, it is important to learn the benefits and drawbacks of related tools to then decide whether to implement an already existing architecture into RIOT OS or to invent a new architecture that can benefit from what was learned while evaluating the work done in competing architectures.

4.3.1 Apache Mynewt Config Subsystem

Advantages

- **Configuration Schemas can have a deeply nested tree structure**
  (see section 3.4)
  The configuration parameter identifier of each handler is stored as a simple string key. This makes it possible to easily have implicit grouping by just defining a long string with multiple separators. Then internally common groups of multiple parameters can be processed by the configuration subsystem.

- **Low implementation effort for modules/drivers**
  (see section 3.9)
  Every module just implements a set and get function inside a handler, that has gets or returns values as strings based on the input string, which is the identifier of a configuration parameter or group.

- **Easy integration with some other external Configuration Managers**
  (see section 3.10)
  Some tools that can be used for external Configuration Managers such as CoAP and MQTT for example can just reuse the configuration parameter path strings as address or topic. They can be written once and don't need to be implemented for every module (handler) of the configuration subsystem.

- **Transactionally Commit Configuration Changes**
  (see section 3.6)
  Mynewt Config can transactionally commit multiple configuration changes at once my calling the "commit" function (see subsubsection 4.2.1).

- **Persistent Configurations**
  (see section 3.6)
  Mynewt Config can persistently save configuration values to a non-volatile storage device by calling the "conf_save" function (see subsubsection 4.2.1).

Disadvantages

- **Configuration Schemas are defined per module/driver**
  (see section 3.1)
  There are no shared configuration structure definitions that can be implemented by different modules. Each module implements its own custom configuration structure. Even if there are 2 different LED drivers, that do exactly the same except they use different hardware, they have no shared code.

- **No support for multiple instances**
  (see section 3.2)
  The subsystem itself has no construct for how instances might work. The implementing module/driver can of course hack around this by implementing an instance

group as part of the path inside the handler and then internally map it to the duplicated devices that shall be configured. But this is a custom solution that will only make the configuration subsystem unnecessarily complex and inconsistent.

- **Path to configuration parameters as string**
  (see section 3.3)
  The identifier of a configuration parameter is a string, which supports nested groups via "/" separators. In general, the structure of this string can be totally custom per module/driver, as the modules that implement a configuration handler decide how to interpret its structure. This causes a lot of overhead in string deserialization and is easy to cause errors or undocumented inconsistencies.

- **Configuration parameters have no type information**
  (see section 3.5)
  Everything is based on strings. The handler implemented by the module must convert the input string into whatever format it needs. And also the other way around on return. In that sense, a type is not needed, as the handler takes care of it with the cost of performance overhead and a more complicated API for the user. If there are types the user could for example know if a number is needed or a string, which can be unclear in some scenarios. But this is not the case with this configuration subsystem.

- **Configuration parameter values are stored in the string format**
  (see section 3.6)
  As every parameter value is returned and set as a string value, it also is necessary to persist it as a string, so that the handler could understand it when it gets read and passed to it. As strings in most cases use up more storage than primitive values such as numbers, this can be a problem for constrained devices with not a lot of memory.

- **Difficult or almost impossible integration with some other external Configuration Managers**
  (see section 3.10)
  External Configuration Managers such as LwM2M have their predefined CS structure called object models. There is for example an object model for a colored light bulb. If this external Configuration Manager is integrated with the Mynewt Config, it is necessary to write a mapping for each Mynewt Config handler to the corresponding object model in LwM2M. Considering that LwM2M is not the only

standard for configuration management this becomes an issue to maintain. Also, some other simpler Configuration Managers could depend on integer arrays as an identifier for the configuration parameter. Transforming a string as is used in the parameter of the Mynewt Config, will always have collisions and is therefore not reliable.

Conclusion

What is needed is not a registry that is split into modules/drivers, but a registry that defines data structures that can be implemented by multiple modules/drivers to share the same interface for identical use cases.

4.3.2 LwM2M Object and Resource Registry

Advantages

- **Configuration Schemas are shared between modules/drivers**
  (see section 3.1)
  LwM2M has predefined CSs for common use cases such as WLAN configuration [5] or location data [5, p. 124] called object models. Modules/drivers can implement these object models and in this way share the same data structure across similar use-cases.
- **Multiple Schema Instances**
  (see section 3.2)
  Each LwM2M object model can be implemented and instantiated by more than one model/driver. To identify different instances, the instance_id is part of the integer path to access configuration parameters: Object model/instance/parameter.
- **Path to configuration parameter as array of 3 integers**
  (see section 3.3)
  The path to identify a single configuration parameter consists of 3 integers: Object Model ID / Instance ID / Parameter ID. This way the payload for requests will not be increased only because an object model has a long name.

- **Configuration parameters have type information**
  (see section 3.5)
  A configuration parameter can have multiple types: String, (Unsigned) Integer, Float, Boolean, Opaque, Time [5, p. 99]. This helps the user to find out what kind of value is valid for a given configuration parameter.
- **Configuration parameter values can have any internal type (string, int, binary, etc.)**
  (see section 3.6)
  The values of configuration parameters get set in their primary type and not formatted in some higher type such as a "string type" for example.
- **Transactionally Commit Configuration Changes**
  (see section 3.6)
  LwM2M has the ability to set multiple configuration parameters at once. If they are set at once, they would also be committed at the same time.
- **Persistent Configurations**
  (see section 3.6)
  The open source LwM2M client implementation called Eclipse Wakaama [24] does not come with this feature, but it is possible to implement this functionality on top of it.

Disadvantages

- **Configuration Schemas can not have a deeply nested tree structure**
  (see section 3.4)
  More complicated object models with many configuration parameters that can be logically structured, will have some unnecessary overhead. A structure can still be simulated by giving names such as: "group_a/group_b/param_c" as parameter names.
- **High implementation effort for modules/drivers**
  (see section 3.9)
  The open source LwM2M client implementation called Eclipse Wakaama [24] has a high implementation effort for module/driver developers to integrate it. For instance, the example implementation in the Wakaama Repository of the LwM2M

location object model [5, p. 125], which only exposes 7 values(latitude, longitude, altitude, radius, velocity, timestamp and speed), already needs 352 lines of code [5, p. 125].

- **Difficult to integrate with other external Configuration Managers**
  (see section 3.10)
  LwM2M is itself an external Configuration Manager and not intended as a middleware that could be integrated with other external Configuration Manager tools. As a consequence, even though possible, the integration of other configuration tools with for example the "Eclipse Wakaama" client [24], which already requires considerably a lot of work to integrate itself as explained earlier, is not easily done. Most importantly to create good external Configuration Manager integrations it is important to use meta-fields such as "name", or "description". Those fields exist in the LwM2M Object Model Definitions [5, p. 68], but these are only known by the LwM2M Server and not part of the LwM2M client running on the RIOT OS node [5]. This way the client cannot expose any human-readable information of its API, except integer paths and types.

Conclusion

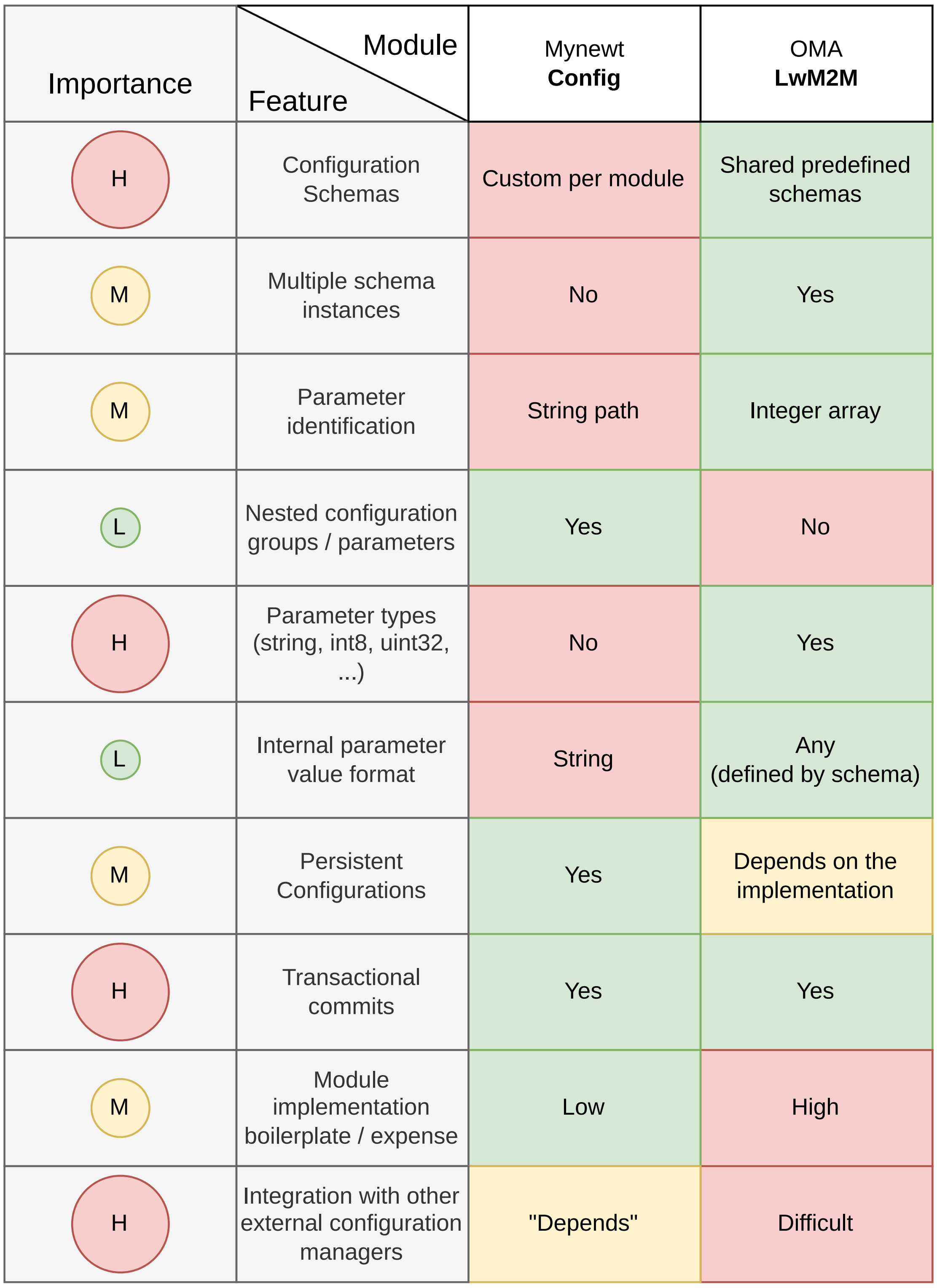
Other Configuration Managers should be supported also. LwM2M is difficult to integrate with other managers. An interface between RIOT OS and LwM2M is needed. For example, a new system module called RIOT OS Registry.

4.4 Summary of Implementation Work Assessment

As Figure 4.1 shows, there is a lot to be learned from existing technologies. Especially in how different their approaches to fixing similar issues are. But not only Mynewt Config but also OMA LwM2M both on their own do not satisfy the needs of a RIOT OS-wide registry good enough, to be implemented as a solution to the problem.

The main issue with Mynewt Config is the fact that each module would need to implement the CS on its own, which results in many similar but not identical CSs inside similar modules/drivers and also prevents the integration of external Configuration Managers such as LwM2M (see subsubsection 4.3.1). Also, the lack of type information for configuration parameters makes the integration of external Configuration Managers that rely on type information problematic.

On the other hand, the main issues with OMA LwM2M are a way too large module implementation boilerplate and the high difficulty to integrate other external configuration managers with the local LwM2M schema.



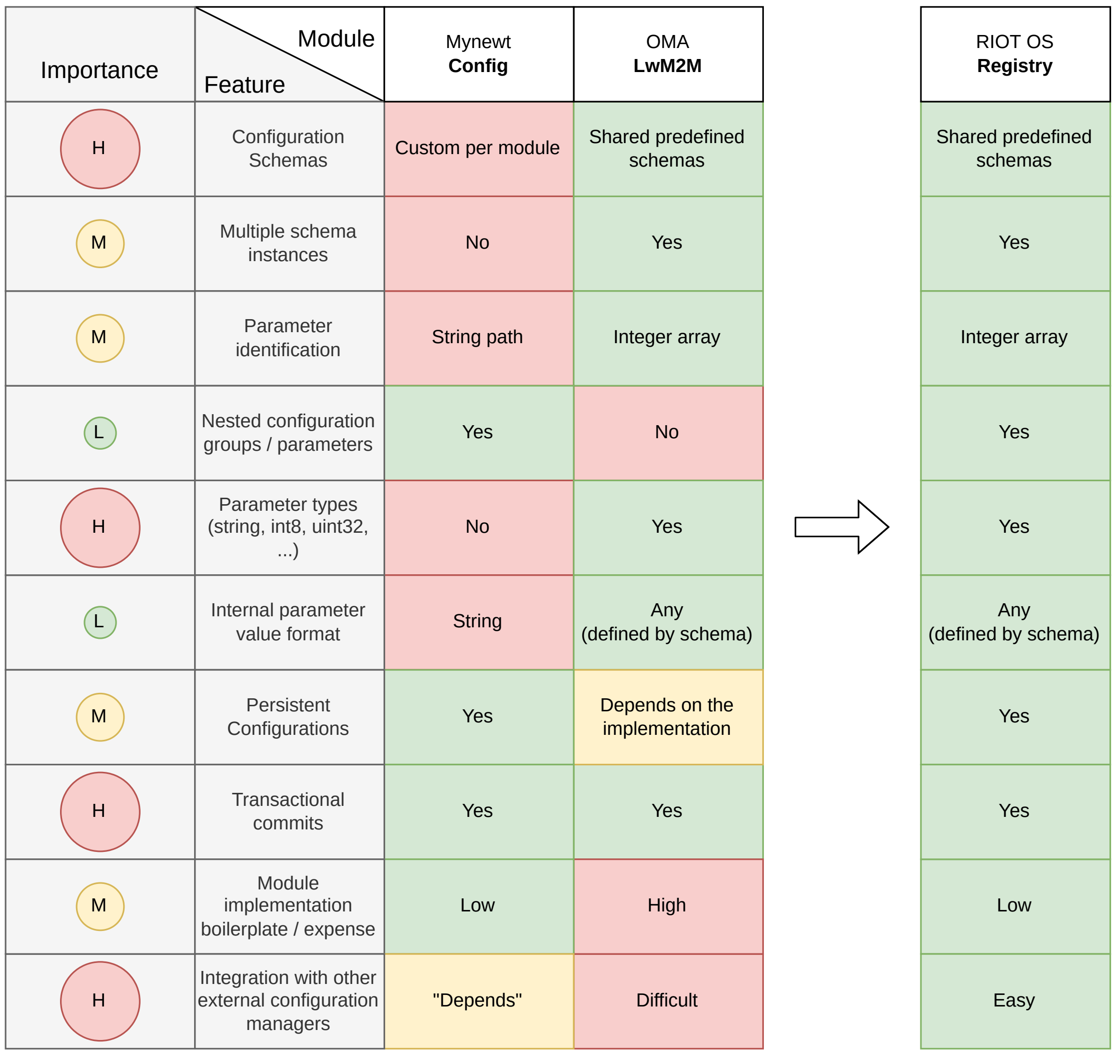
Importance	Module / Feature	Mynewt **Config**	OMA **LwM2M**
H	Configuration Schemas	Custom per module	Shared predefined schemas
M	Multiple schema instances	No	Yes
M	Parameter identification	String path	Integer array
L	Nested configuration groups / parameters	Yes	No
H	Parameter types (string, int8, uint32, ...)	No	Yes
L	Internal parameter value format	String	Any (defined by schema)
M	Persistent Configurations	Yes	Depends on the implementation
H	Transactional commits	Yes	Yes
M	Module implementation boilerplate / expense	Low	High
H	Integration with other external configuration managers	"Depends"	Difficult

Figure 4.1: Related work influences.

4.5 Conclusion of Implementation Work Assessment

The logical consequence is the creation of a new configuration registry that is based on the concepts of Mynewt Config and LwM2M but only uses those that fit the needs of the RIOT OS most. It is supposed to keep the simplicity of the Mynewt Config, by also supporting the advantages of OMA LwM2M. The OMA LwM2M configuration manager can then be implemented through a mapping between the new RIOT OS Registry configuration parameters and the LwM2M object models.

As can be seen on the right-hand side of Figure 4.2. This new RIOT OS Registry allows the specification of CSs that are shared by drivers and modules so that every driver or module that represents the same functionality will also have the same structural representation of their configuration parameters. This also easily allows the ability to create multiple instances of the same CS. For example, a traffic light would need 3 LEDs that use the same CS. The identification of a configuration parameter of the RIOT OS Registry is a result of its path, which is an array of integers, of which the length depends on how deeply nested the CS structure is. To improve the integration of typed configuration managers, the configuration parameters also have metadata containing type information, but also strings such as “name” or “description” to allow the creation of more simple APIs for developers. For example a Command-Line Interface (CLI) that allows using the name field as an alias to the integer array. Internally the RIOT OS Registry allows the CS to use every available type in the C programming language to specify the value of a configuration manager that is written to the program storage. To prevent unnecessary conversion from string to native value and the other way around. The integration of the new registry into drivers or modules is also supposed to be as simple as possible. Therefore, an API that is inspired by the Mynewt Config (see subsection 4.2.1) will be implemented.



Importance	Feature \ Module	Mynewt **Config**	OMA **LwM2M**		RIOT OS **Registry**
H	Configuration Schemas	Custom per module	Shared predefined schemas		Shared predefined schemas
M	Multiple schema instances	No	Yes		Yes
M	Parameter identification	String path	Integer array		Integer array
L	Nested configuration groups / parameters	Yes	No		Yes
H	Parameter types (string, int8, uint32, ...)	No	Yes	⇨	Yes
L	Internal parameter value format	String	Any (defined by schema)		Any (defined by schema)
M	Persistent Configurations	Yes	Depends on the implementation		Yes
H	Transactional commits	Yes	Yes		Yes
M	Module implementation boilerplate / expense	Low	High		Low
H	Integration with other external configuration managers	"Depends"	Difficult		Easy

Figure 4.2: Related work conclusion.

5 Design of the new RIOT OS RCS

5.1 Architecture

The proposed RCS architecture, as shown in Figure 5.1, is formed by one or more Configuration Managers (see section 5.3) and the RIOT OS Registry (see section 5.2). The RIOT OS Registry acts as a common interface to access Runtime Configurations and store them in non-volatile devices. All runtime configurations can be accessed either from the RIOT OS application or the interfaces exposed by the Configuration Managers, via the RIOT OS Registry. A RIOT OS Application may interact with a Configuration Manager in order to modify access control rules or enable different exposed interfaces.

Figure 5.1 shows this in more detail. It differentiates between 2 different kinds of Configuration Managers:

Basic Configuration Managers:

These Configuration Managers are a simple representation of the default configuration structure of the RIOT OS Registry. They only expose the parameters paths as is and do not map to any special structure.

Advanced Configuration Managers:

These Configuration Managers have their own configuration structure (custom predefined object models etc.) and can not automatically be mapped to from the RIOT OS Registry itself. To make them work, a custom mapping module needs to be implemented, which maps each configuration parameter from the registry to the correct format of the Configuration Manager.

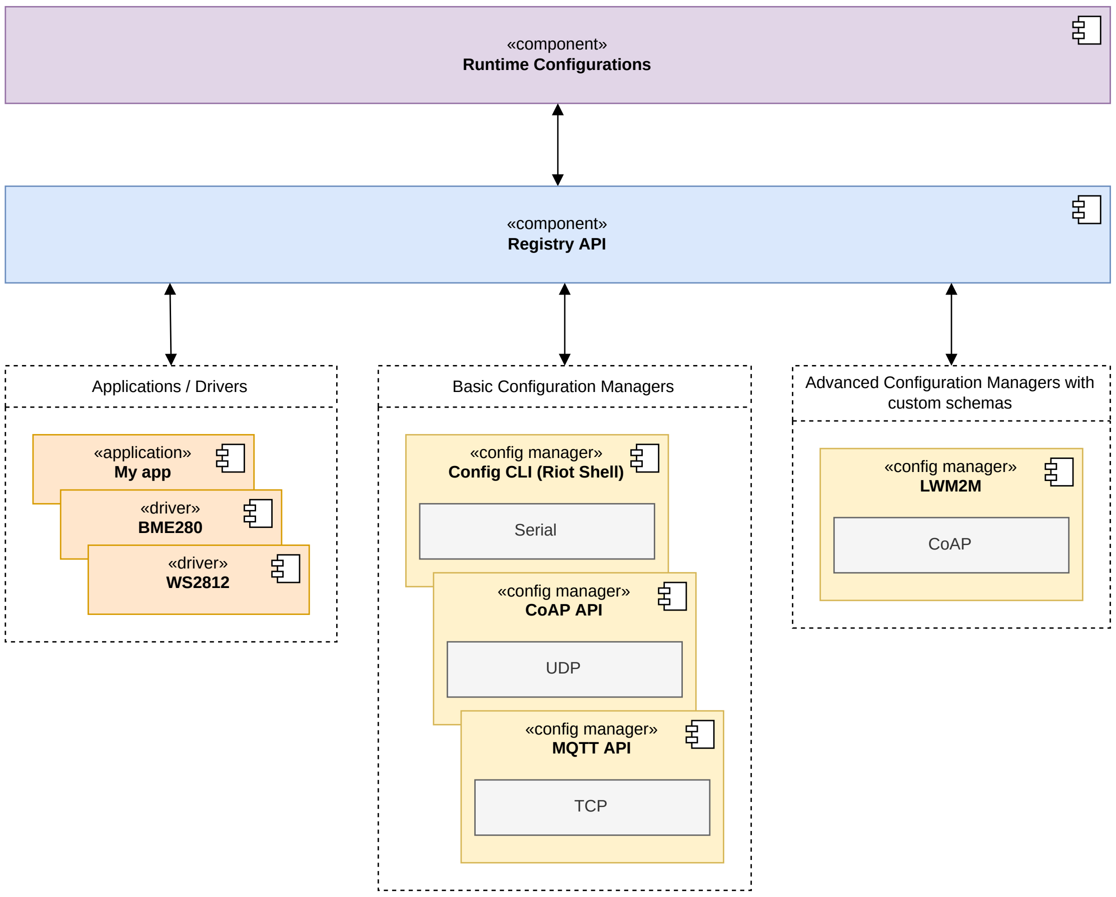


Figure 5.1: Runtime Configuration Architecture.

5.2 RIOT OS Registry

The RIOT OS Registry is a module for interacting with persistent key-value configurations. It's heavily inspired by the Mynewt Config implementation and LwM2M Object Models [5, p. 68].

The RIOT OS Registry interacts with RIOT OS modules via CSs (see subsection 5.2.1), and with non-volatile storage devices via Storage Facilities (see subsection 5.2.2). This way the functionality of the RIOT OS Registry is independent of the functionality of a module or storage device. It is possible to get or set the values of configuration parameters. A CP is used to point to the correct configuration parameter. It is also possible to transactionally apply configurations or export their values to a buffer or print them. To persist configuration values, it is possible to store them in non-volatile storage devices.

Any mechanism of security (access control, encryption of configurations) is not directly in the scope of the Registry but in the Configuration Managers and the specific implementations of the CS and SF.

Figure 5.2 shows an example of two CSs (My app, LED Strip). The application "My app" uses the custom "My app" CS to expose custom configuration parameters to the RIOT OS Registry and the drivers WS2812, SK6812 and UCS1903 contain instances of the "LED Strip" CS to expose common LED Strip configuration parameters. Also, there are two Storage Facilities available: EEPROM and FAT.

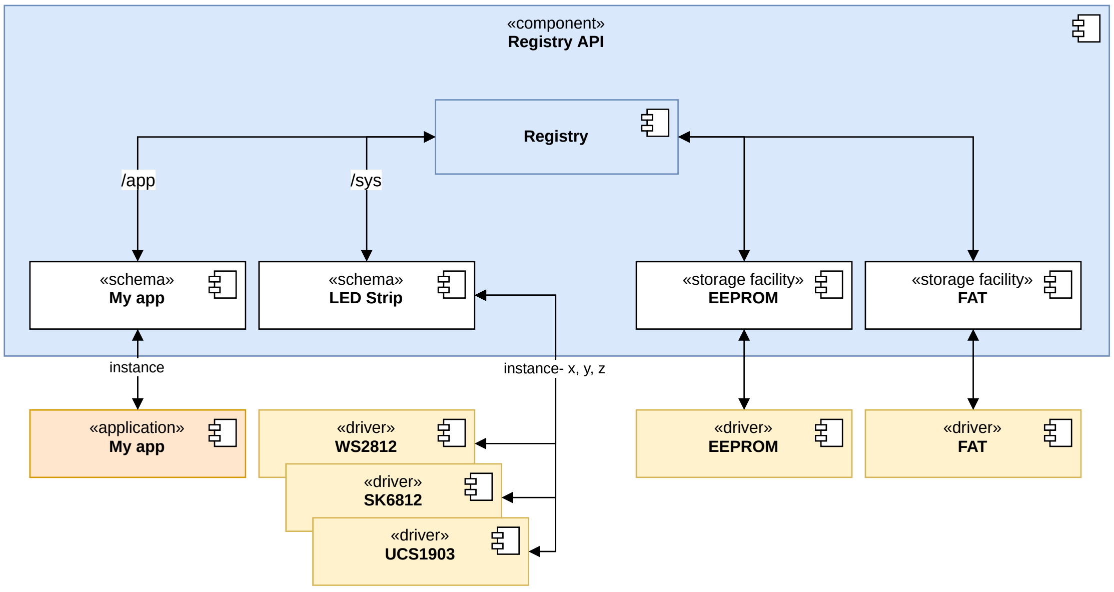


Figure 5.2: The RIOT OS Registry components.

See Usage Flow (subsection 5.2.4) for more information.

5.2.1 Configuration Schema (CS)

A CS represents a CG in the RIOT OS Registry. A RIOT OS module is required to add an instance to a given CS in order to expose its configurations to the Registry API. Or needs to implement its own custom CS.

A CS is defined by an ID, some metadata (name, description) and a get and set handler for interacting with the configuration parameters of the CG.

- set: Sets a value to a configuration parameter.
- get: Gets the current value of a configuration parameter.

The CS also contains the struct that specifies how each instance (SI) stores the actual data.

Schema Instance (SI)

An instance of a CS, which contains the actual data values. It can be added to a CS and contains a "commit_cb" handler, to notify the module containing the instance about configuration changes that need to be applied.

- commit_cb: To be called once configuration parameters have been set, in order to apply any further logic required to make them effective (e.g. handling dependencies).

5.2.2 Storage Facility (SF)

An SF must implement the "storage interface" to allow the RIOT OS Registry to load, search and store configuration parameters. From the point of view of the RIOT OS Registry, all parameters are key/value pairs with certain types, it is the responsibility of the SF to transform those into a proper format to store them. (E.g. lines separated by a "\n" character in a file or encoded in CBOR etc.).

The interface of an SF is defined with a descriptor that has the following attributes:

- load: Executes a callback function for every configuration parameter stored in the storage.
- store: Stores one configuration parameter in the storage.

Any kind of storage encryption mechanism is not in the scope of this document, and up to the implementation of load and store or intrinsic encryption functionalities in the storage.

A minimal RIOT OS Registry setup requires at least one source SF from which configurations are loaded and exactly one SF destination to which configurations are stored. Having multiple SF sources can be useful when it's required to migrate the data between Storage Facilities (e.g to migrate all configurations from SF A to B, register B as source and destination and add A as a source).

5.2.3 Configuration Path (CP)

A complete CP is a unique identifier of a configuration parameter. A CP does not need to be complete and can also only point to a specific Configuration Namespace (CN), CS, SI or CG. The RIOT OS Registry needs this information, so that it knows where to look for the requested configuration parameter values or metadata. Below is a regex example showing how the CP is structured. All path elements have to be integers: "namespace_id/schema_id/instance_id/(group_id/)*parameter_id". In reality the amount of "group_ids" is limited to 8 and can be changed with a 'define', so the regex is a bit simplified.

Configuration Namespace (CN)

A CN splits CSs in multiple categories. Currently specified are the following: "SYS=0" and "APP=1". CSs that are part of "SYS" are RIOT OS internal CSs and are used to abstract common configuration structures within RIOT OS such as "IEEE802154" etc. The "APP" CN must not be used by RIOT OS itself, but only by the application. This is to prevent application specific CS from clashing with RIOT OS's internal CS. This is specifically important for the case of when new CSs are added in a future RIOT OS version.

Configuration Group (CG)

Within RIOT, each SI contains a list of configuration parameters and/or CGs. A CG can contain multiple sub-CGs. This way a more complex CS can be split into multiple

CGs, logically separating configuration parameters, instead of having them all in a flat key-value list. Because the RIOT OS Registry allows a CP to point to specific CGs, this gives the ability to do operations on a set of configuration parameters that share the same CG, without needing to address each of those configuration parameters separately.

5.2.4 API and Usage Flows

API

Figure 5.3 shows the API of the RIOT OS Registry. On the left-hand side the basic API to manage configuration parameters is shown. It allows to "set" and "get" configuration parameters, transactionally "commit" them, "export" them to a buffer or terminal, "load" them from storage and to "save" them to the storage. On the right-hand side the setup API is shown, exposing functions to register CSs, SIs and SFs.

The functionality is of these functions is explained in the following paragraphs.

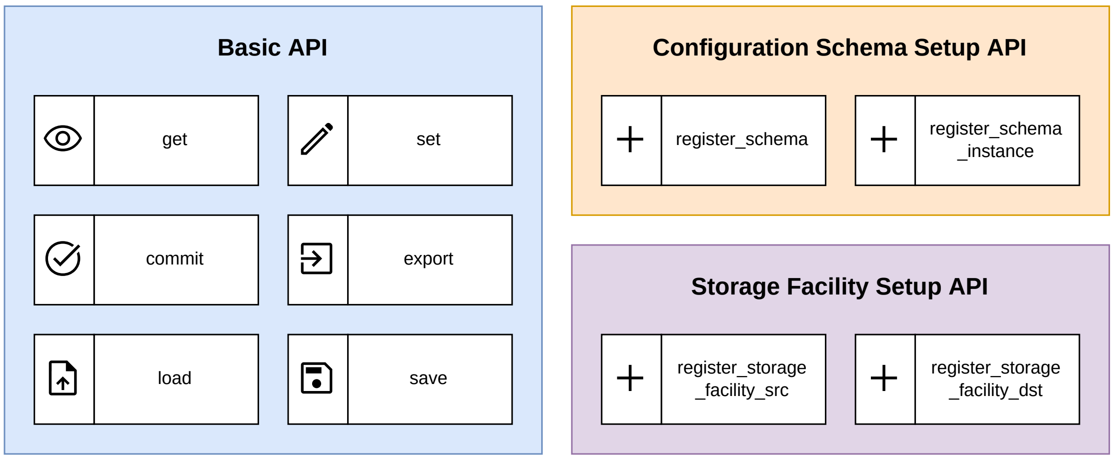

Basic API
get
set
commit
export
load
save
Configuration Schema Setup API
register_schema
register_schema _instance
Storage Facility Setup API
register_storage _facility_src
register_storage _facility_dst

Figure 5.3: RIOT OS Registry API.

Registry Initialization

As described in the flow in Figure 5.4, modules add their SIs to predefined CSs or declare and register their own CS for CGs in the RIOT OS Registry. SFs are registered as sources and/or destinations of configurations in the RIOT OS Registry.

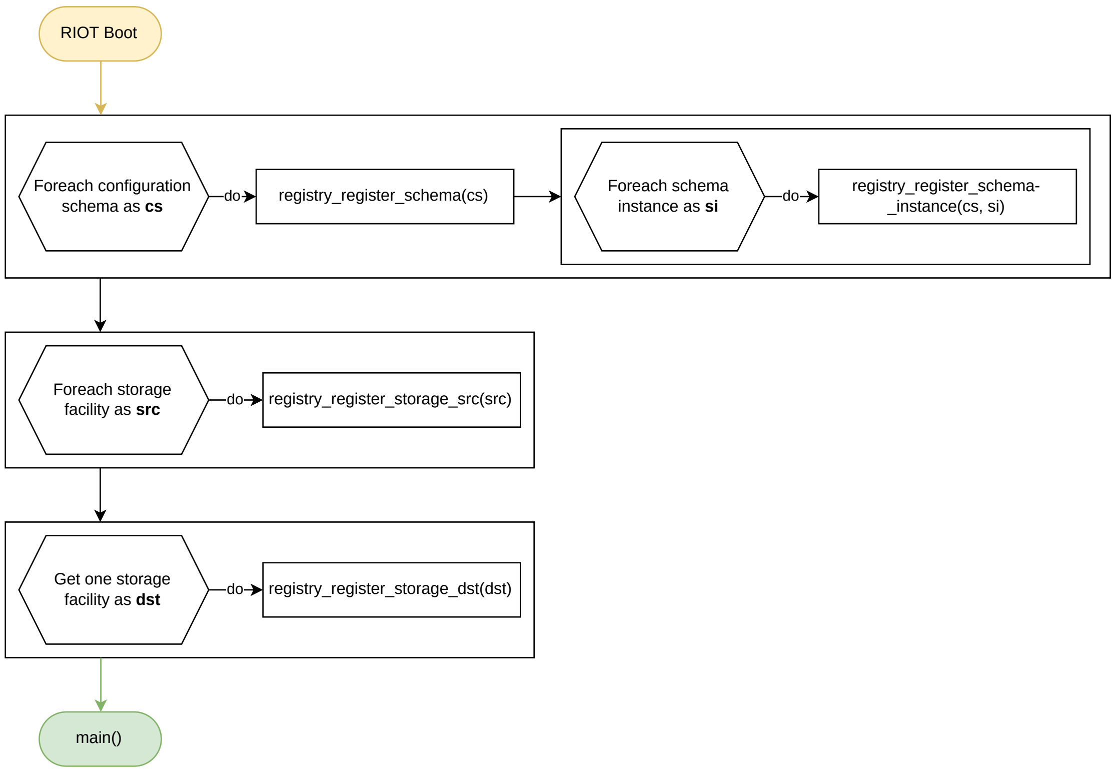

RIOT Boot
Foreach configuration schema as **cs**
do
registry_register_schema(cs)
Foreach schema instance as **si**
do
registry_register_schema-_instance(cs, si)
Foreach storage facility as **src**
do
registry_register_storage_src(src)
Get one storage facility as **dst**
do
registry_register_storage_dst(dst)
main()

Figure 5.4: Usage flow of the RIOT OS Registry.

Get Configurations

At any time, the application or a Configuration Manager can retrieve a configuration value using the (registry_get_value) function.

Figure 5.5 shows the flow of getting the value of a configuration parameter. First the function "registry_get_value" is called and takes the CP as its argument. If the registry can find the requested CN, CS, SI and optionally all the CGs that are part of the CP and if the last element of the CP is a configuration parameter, then it gets its value from the SI and returns it. Otherwise the error "ENOTFOUND" is returned.

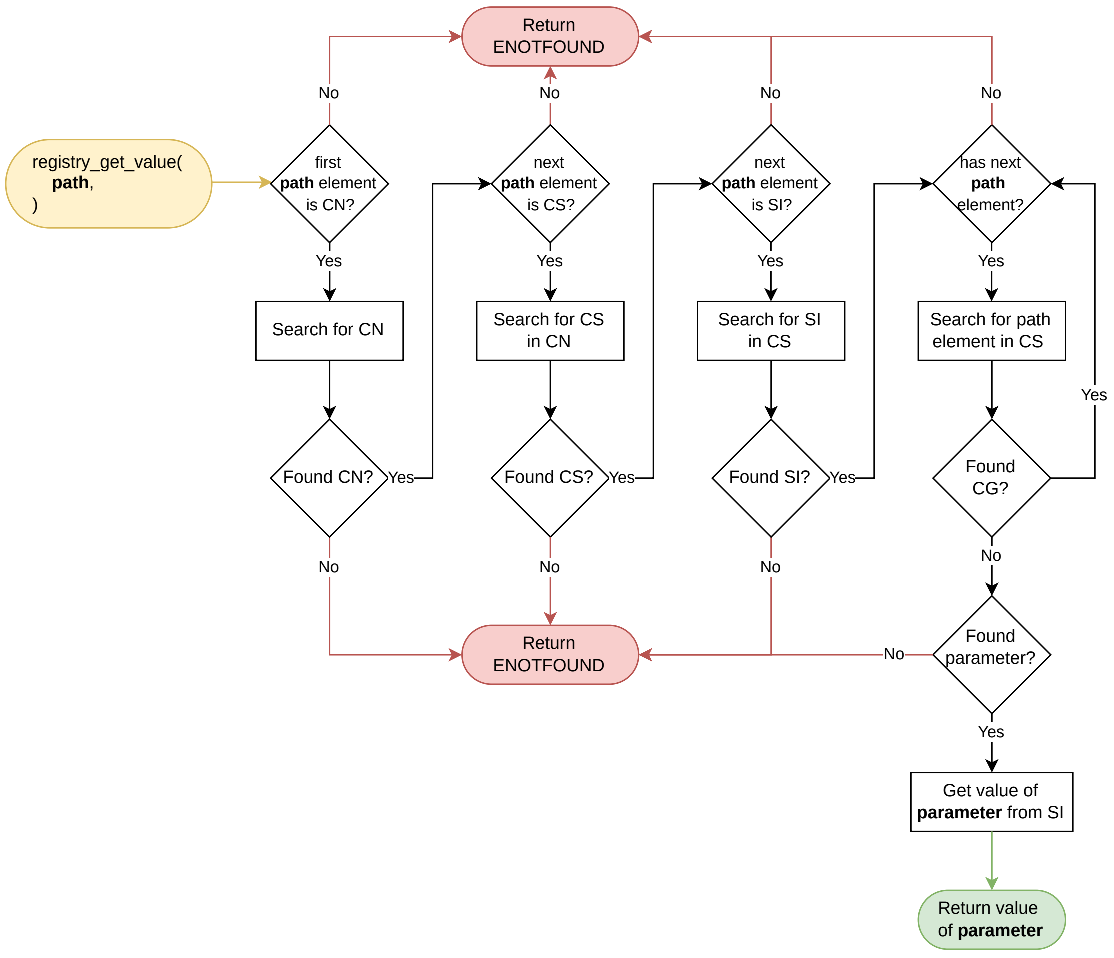


Figure 5.5: Behavioral flow of the "get" function.

Set Configurations

At any time, the application or a Configuration Manager can set a configuration value using the (registry_set_value) function.

Figure 5.6 shows the flow of setting a configuration parameter to a new value. First the function “registry_set_value” is called and takes the CP as its argument. If the registry can find the requested CN, CS, SI and optionally all the CGs that are part of the CP and if the last element of the CP is a configuration parameter, then it sets its value inside the SI to the new value. Otherwise the error “ENOTFOUND” is returned.

Note this function doesn't interact with the SF, so configuration changes are not reflected in the non-volatile storage devices unless the function “registry_save” is called (see subsubsection 5.2.4).

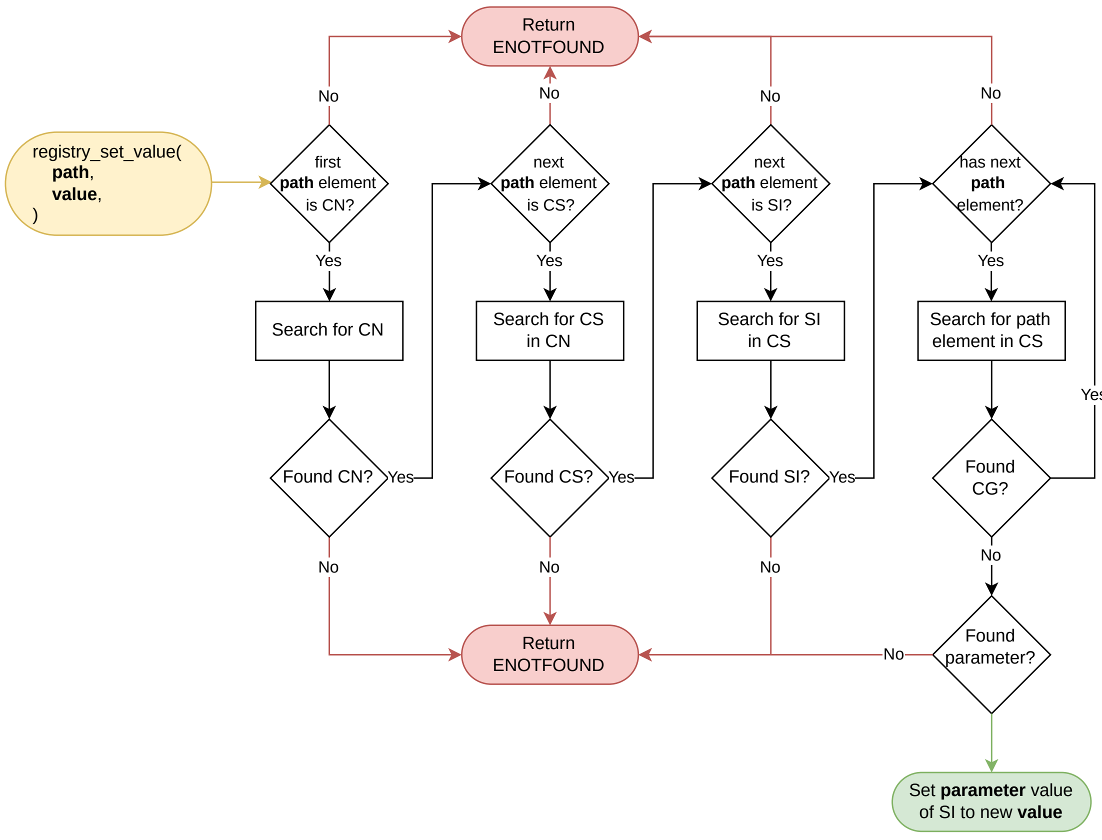


Figure 5.6: Behavioral flow of the "set" function.

Commit Configurations

Once the value(s) of one or multiple configuration parameter(s) are changed by the “registry_set” function, they still need to be committed, so that the new values are taken into effect. At any time, the application or a Configuration Manager can commit a specific, or multiple configuration value(s) using the (registry_commit) function.

Figure 5.7 shows this process in more detail: First, the function “registry_commit” is called and takes a CP as its argument. If the registry can find the requested CP, each configuration parameter within this given CP will be passed on to the “commit_cb” handler of the SI, taking its full CP as an argument. This callback is implemented by the modules/drivers that own the SI of the currently called configuration parameter. This way they get notified, when the configuration parameter has been committed and can apply the changes accordingly.

If the registry does not find parts of the given CP, it returns a “ENOTFOUND” error.

If the given CP does not point all the way to a concrete configuration parameter, but only to a CN, a CS, a SI or a CG, then the registry will search for the configuration parameters of all the children of the specified CP recursively and commit them.

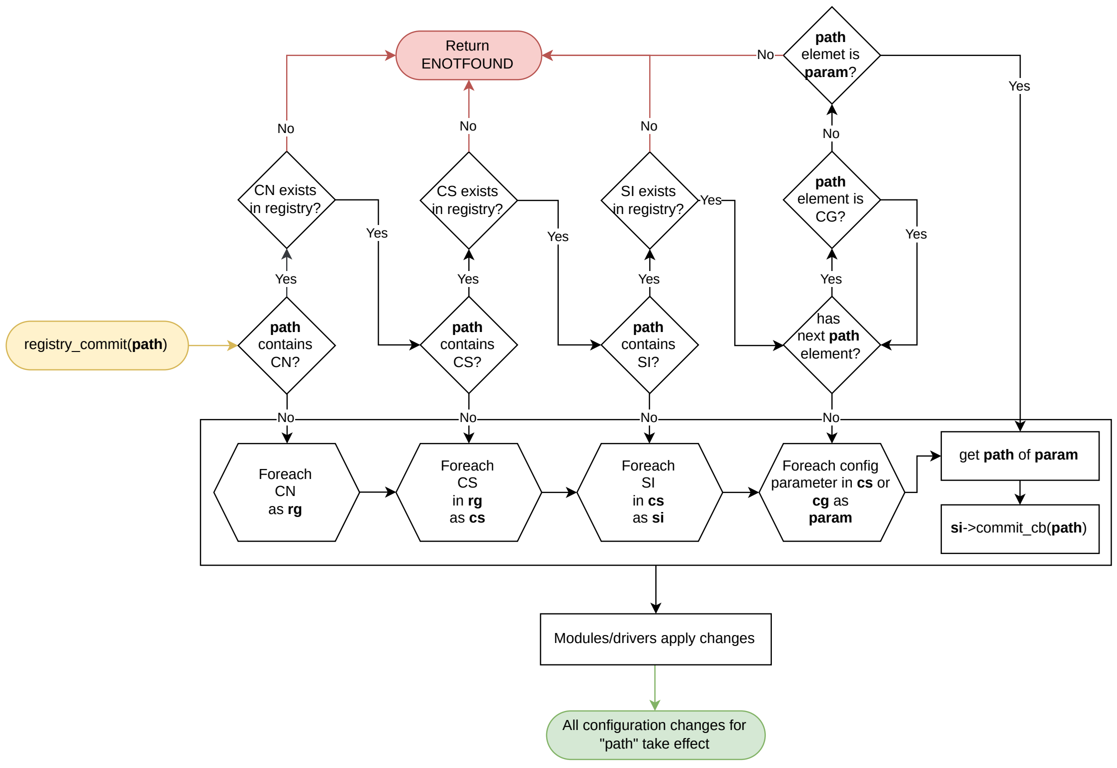


Figure 5.7: Behavioral flow of the "commit" function.

Export Configurations

At any time, the application or a Configuration Manager can export a specific, or multiple configuration value(s) using the (registry_export) function.

Figure 5.8 shows the flow of exporting values and metadata of configuration parameter that are children of the given CP. First, the function “registry_export” is called and takes a CP and a “export_func” callback as argument. If the registry can find the requested CP, each configuration parameter within this given CP will be passed on to the “export_-func” callback, taking itself and its value as an argument, in this way exporting the configuration parameters.

If the registry does not find parts of the given CP, it returns a “ENOTFOUND” error.

If the given CP does not point all the way to a concrete configuration parameter, but only to a CN, a CS, a SI or a CG, then the registry will search for the configuration parameters of all the children of the specified CP recursively and export them.

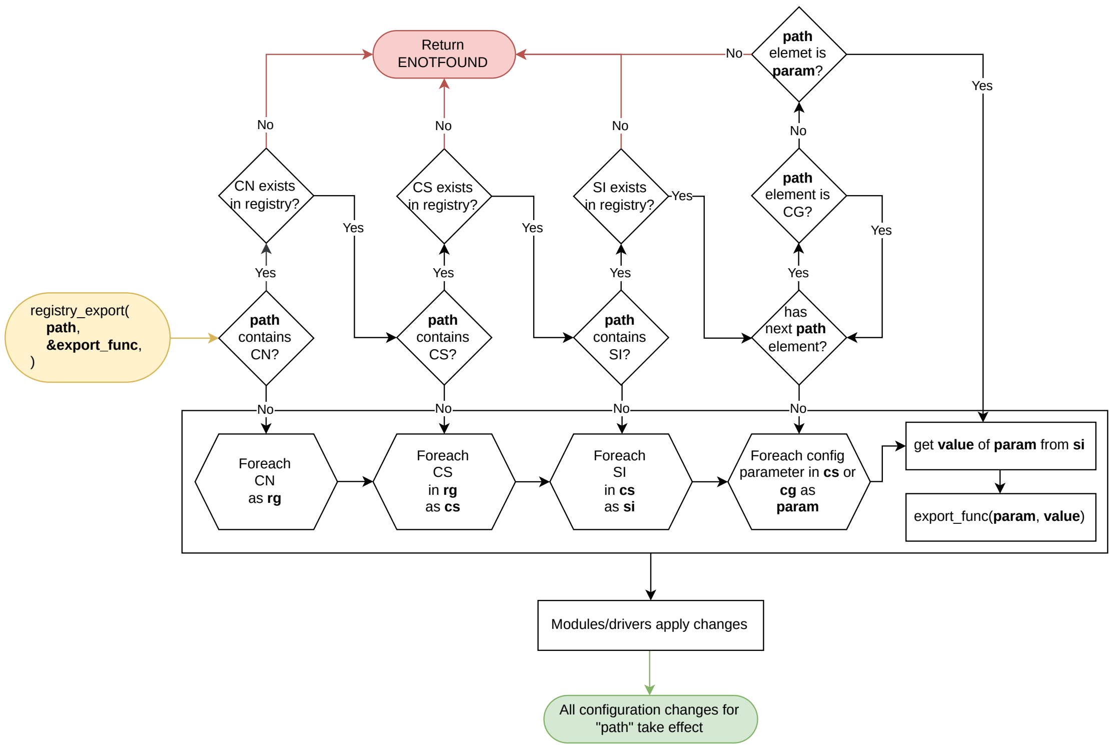


Figure 5.8: Behavioral flow of the "export" function.

Load Configurations from Storage

At any time, the application or a Configuration Manager can load all configurations from the registered SF sources (registry_load function). For example when a device restarts after a shutdown.

Figure 5.9 shows this process in more detail: First, the “registry_load” function is called with a CP as argument, specifying which configuration parameters must be loaded from storage. Then the “registry_load” function internally calls the SF’s “load” handler with the storage instance, CP and the “load_func” callback, which is set to the “registry_-set_value” function, as arguments. Then the SF calls the “registry_set_value” function for each configuration parameter that it finds on the storage instance’s storage device.

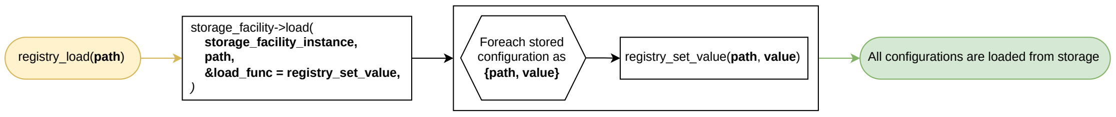

registry_load(path)
storage_facility->load(storage_facility_instance, path, &load_func = registry_set_value,)
Foreach stored configuration as {path, value}
registry_set_value(path, value)
All configurations are loaded from storage

Figure 5.9: Behavioral flow of the “load” function.

Save Configurations to Storage

At any time, the application or a Configuration Manager can store all configurations in the SF destination (registry_save function). For example to prevent configuration loss in case of a shutdown of the device.

Figure 5.10 shows this process in more detail: First, the “registry_save” function is called with a CP as argument, specifying which configuration parameters must be saved to storage. Then the “registry_save” function internally calls the “registry_export” function with the CP and the SF’s “save” handler, as arguments. Using the SF’s “save” handler as the export handler of the “registry_export” function, causes the “registry_export” function to call it for each configuration parameter, that is within the specified CP and passing its value on with it. Then the SF’s “save” handler each time saves the given configuration parameter to the storage instance’s storage device.

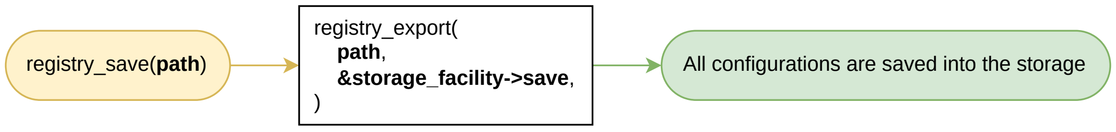

registry_save(path)
registry_export(path, &storage_facility->save,)
All configurations are saved into the storage

Figure 5.10: Behavioral flow of the “save” function.

Add Custom Configuration Schemas to the Registry

The registry itself already comes with many CSs that live within the “sys” CN. But sometimes an application needs some custom runtime configurations that are too specific for the registry to abstract, so it is possible to register a custom CS within the “app” CN.

One must not register a custom CS within the “sys” CN, as this is a reserved space and using it would almost certainly result in conflicts whenever RIOT OS gets updated.

Figure 5.11 visualizes the behavioral flow of adding a custom CS:

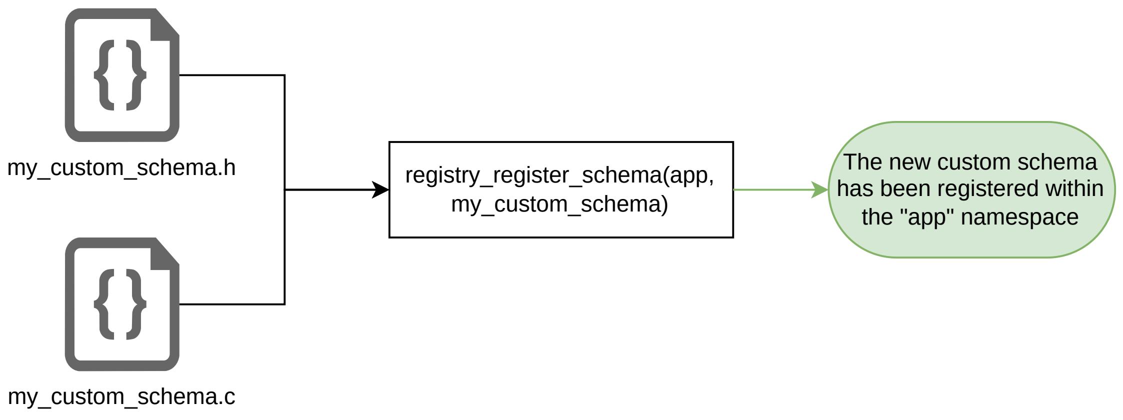

my_custom_schema.h
my_custom_schema.c
registry_register_schema(app, my_custom_schema)
The new custom schema has been registered within the "app" namespace

Figure 5.11: Behavioral flow of the registration of custom CSs.

5.3 Integration of External Configuration Managers

5.3.1 Simple Configuration Managers

Simple Configuration Managers are ways to use the RIOT OS Registry without the need to maintain adapters. Those managers would only be implemented once and mirror the internal structure of the RIOT OS Registry. This can be quite powerful within RIOT OS-only environments, but is not as powerful in terms of its “plug and play” capabilities.

Command-Line Interface (CLI)

The RIOT OS CLI can be extended with a “registry” command, which is followed by a sub-command “set | get | commit | export”.
Each sub-command has a specific CLI interface:

- get: <path>
- set: <path> <value>
- commit: <path>

- export: <path> [-r <recursion depth>]
- load: [path]
- save: [path]

The <path> argument is a string of integers separated by "/". It maps directly to the RIOT OS Registry internal path structure. The <value> argument is just the value as a string. The "export" command also has the additional "-r <recursion depth>" flag. It defaults to 0, which means that everything will be exported recursively. A value of 1 means, that only the parameter that exactly matches the specified path will be exported. A value of 2 means the same as a value of 1 but also all of its children will be exported etc.

CoAP API

The CoAP API based integration uses the RIOT OS internal registry structure and does not come with its own CS structure. But CoAP only has a "get" and "set" function, but no "export" or "commit" function. So the get and set command of the RIOT OS Registry will just be mapped to the get and set of CoAP. For example: "GET /namespace_id/schema_id/..." or "SET /namespace_id/schema_id/...-> new_value". The "export" command can be realized through the "GET /.well-known/core" endpoint. The "commit" command is less trivial as there is no equivalent construct within CoAP itself. But here are some ideas:

- Make a get request which's path has a "commit" prefix such as: "GET /commit/-namespace_id/schema_id/..."
- Have a dedicated "commit" endpoint, which can be set to a specific path, which current state will be committed on execution. For example: "SET /commit -> /namespace_id/schema_id/...".
- Don't implement the "commit concept" at all, but rather commit every "set" operation and allow sending values to whole CGs/CSs as their endpoint, containing values for the complete CG/CS or parts of it. For example in the CBOR or JavaScript Object Notation [3] (JSON) format. This way it still is possible to change multiple values at once.

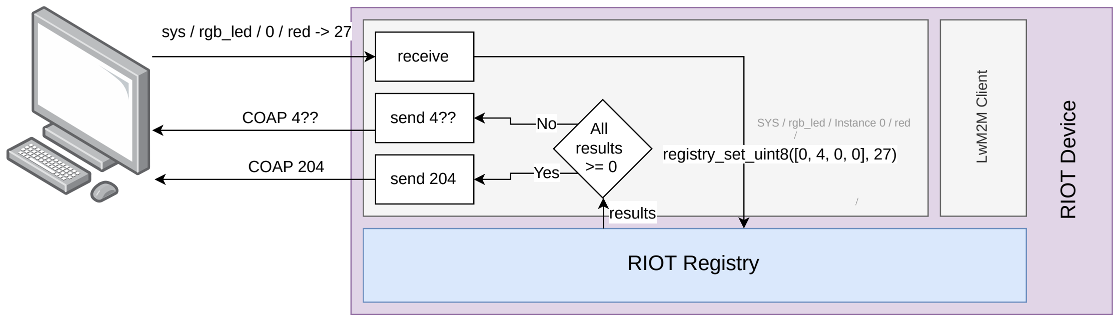


Figure 5.12: CoAP integration.

MQTT API

The MQTT API based integration uses the RIOT OS internal registry structure and does not come with its own schema structure, but is limited to only having events with or without data. As a consequence there are no commands such as set, get, commit or export. Values will be set by sending a "publish" event containing the new value and subscribing to the same event will notify the subscriber whenever a new value is available. This way the "set" and "get" behavior of the RIOT OS Registry can be realized. The export command is not necessary because the MQTT broker gets an initial publish for each parameter when the device boots. So it knows about all existing topics and can expose them. Because one MQTT broker can have multiple RIOT OS nodes, it is necessary to prefix the topic of each message with a device_id. For example: "device_-id/namespace_id/schema_id/...". Less trivial is how the "commit" command can be exposed to MQTT. But here are some ideas:

- Extend the topic of the path that needs to be committed with a "commit" prefix. For example: "commit/device_id/namespace_id/schema_id/...".
- Have a dedicated "commit" topic, which can be set to a specific path, which then will be committed. For example: "SET /commit -> /namespace_id/schema_id/...".
- Don't implement the "commit concept" at all, but rather commit every "set" operation and allow sending values to whole CSs/CSs as their endpoint, containing values for the complete CS/CS or parts of it. For example in the CBOR or JSON format. This way it still is possible to change multiple values at once.

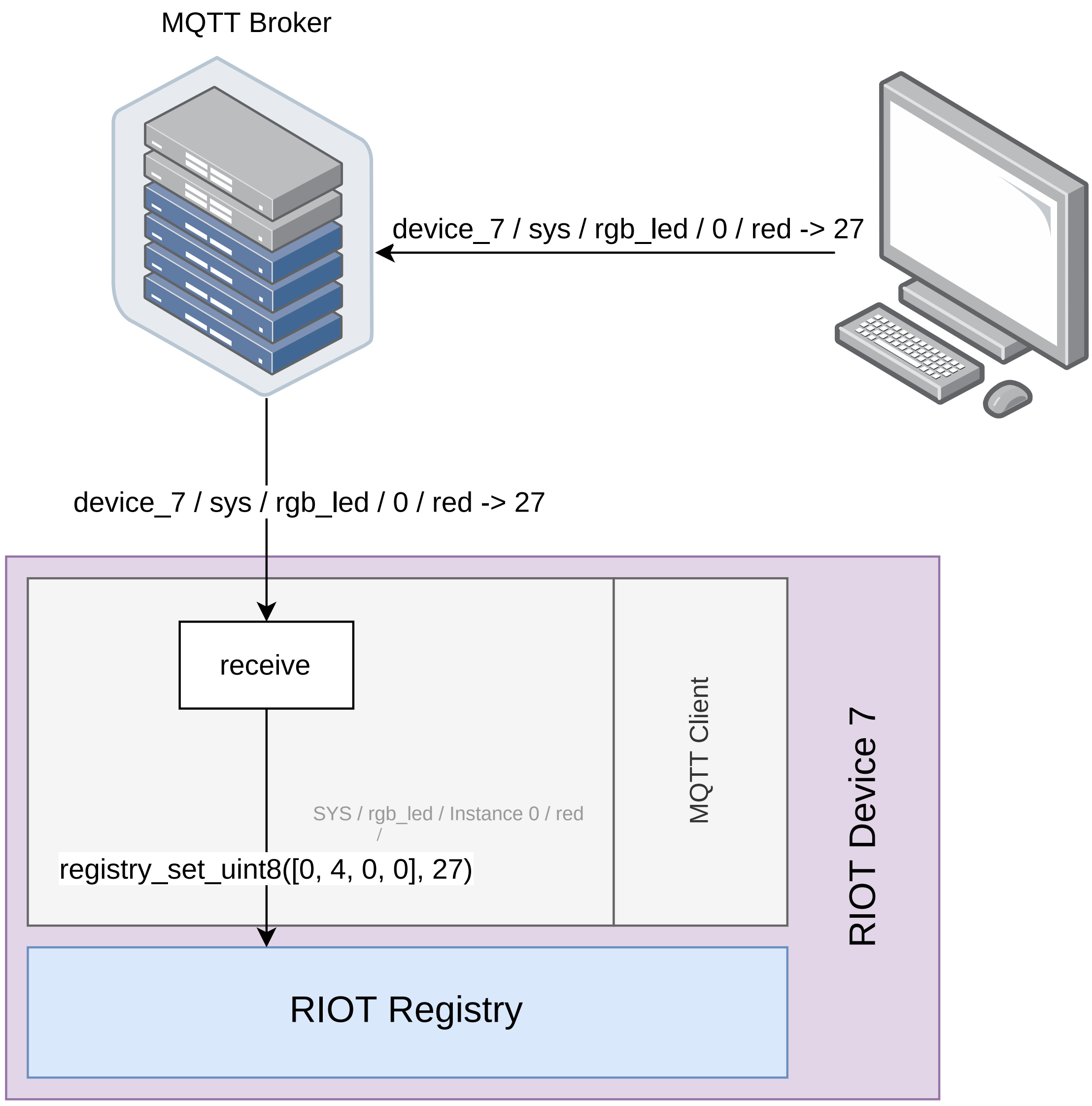

MQTT Broker
device_7 / sys / rgb_led / 0 / red -> 27
device_7 / sys / rgb_led / 0 / red -> 27
receive
SYS / rgb_led / Instance 0 / red /
registry_set_uint8([0, 4, 0, 0], 27)
MQTT Client
RIOT Device 7
RIOT Registry

Figure 5.13: MQTT integration.

5.3.2 Advanced Configuration Managers

While having the ability to use the Registry inside RIOT OS and using a (UART) CLI, the registry itself is designed so that it can easily integrate with common external Configuration Managers. This makes it possible to modify parameters for example via the Ethernet, LoRa, Bluetooth, 802.15.4 etc. The basic idea is that the RIOT OS Registry with its predefined CSs defines a RIOT OS internal specification, as to which kind of data is to be found where. Then each external Configuration Manager has to implement its adapter module, which maps/converts its data structures to the RIOT OS Registry.

LwM2M

LwM2M is a relatively new protocol that is similar to the RIOT OS Registry in that it specifies official (and unofficial) so-called "object models" that define which information can be found where. It internally uses CoAP and has a concept of instances as well. A typical LwM2M configuration parameter identifier/path has the following structure: "object_id/instance_id/parameter_id". The "object_id" is similar to RIOT OS's "schema_id", the "instance_id" is the same as in RIOT OS and the "parameter_id" is also the same as in RIOT OS except LwM2M does not know anything about nesting, so there are no paths longer than '3' [5]. To integrate LwM2M into the RIOT OS Registry it is necessary to write an adapter that maps the LwM2M object models to the RIOT OS Registry CSs. An example of how this adapter would handle a "set" operation can be seen below:

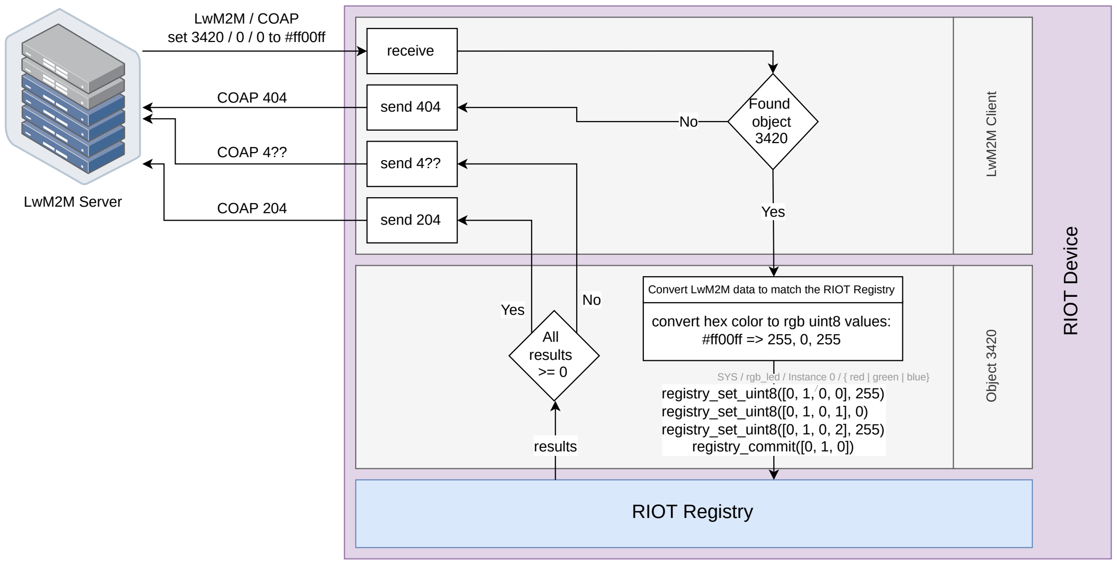

LwM2M / COAP
set 3420 / 0 / 0 to #ff00ff
COAP 404
COAP 4??
COAP 204
LwM2M Server
receive
send 404
send 4??
send 204
No
Found object 3420
Yes
LwM2M Client
RIOT Device
Convert LwM2M data to match the RIOT Registry
convert hex color to rgb uint8 values:
#ff00ff => 255, 0, 255
Yes
No
All results >= 0
SYS / rgb_led / Instance 0 / { red | green | blue}
registry_set_uint8([0, 1, 0, 0], 255)
registry_set_uint8([0, 1, 0, 1], 0)
registry_set_uint8([0, 1, 0, 2], 255)
registry_commit([0, 1, 0])
results
Object 3420
RIOT Registry

Figure 5.14: LwM2M integration.

6 Implementation of the RIOT OS Registry

This chapter shows the implementation details of chapter 5. The full source code is on the enclosed CD and on GitHub [1].

6.1 Configuration Schema (CS)

This section shows the implementation details of the in subsection 5.2.1 specified CSs. The "sys"-CN CSs are implemented in an additional module called "registry_schemas". By default every CS, that is implemented inside that module is disabled, because those implementations are depending on CFLAGS to be set. The structure of those CFLAGS is the following: "DCONFIG_REGISTRY_ENABLE_SCHEMA_{schema_name}", where "schema_name}" must be replaced with the name of an existing CS such as "RGB_LED" or "FULL_EXAMPLE".

Figure 6.1 shows the structure of a CS. It consists of an id, which is used in the CP, a name and description metadata field so that Configuration Managers can provide a less confusing interface, a list of SIs and an items field, which contains an array of "Schema Items". The list of SIs needs to be stored in the CS because the RIOT OS Registry is supposed to do avoid dynamic heap allocation (see section 3.6). This list is a linked list, so every newly registered SI will be added at the end of the list. The Schema Items of the items field represent either a configuration parameter or a CG. Each Schema Item has a Registry ID as well as a name and a description metadata field. If a Schema Item is a CG, then it contains an array of Schema items as its value union field. Otherwise it contains a configuration parameter type of the concrete type "Registry Type".

[1] `https://github.com/LasseRosenow/riot-runtime-config`

Besides its fields the CS also contains a callback function inside its “mapping” field. This function translates a configuration parameter ID to a pointer of the configuration parameter’s value inside the SI. It also returns the size of this configuration parameter value. This function is necessary because how a SI stores its data is decided by each CS’s implementation. So only the CS knows how to translate between a configuration parameter ID and the actual data location inside the SI.

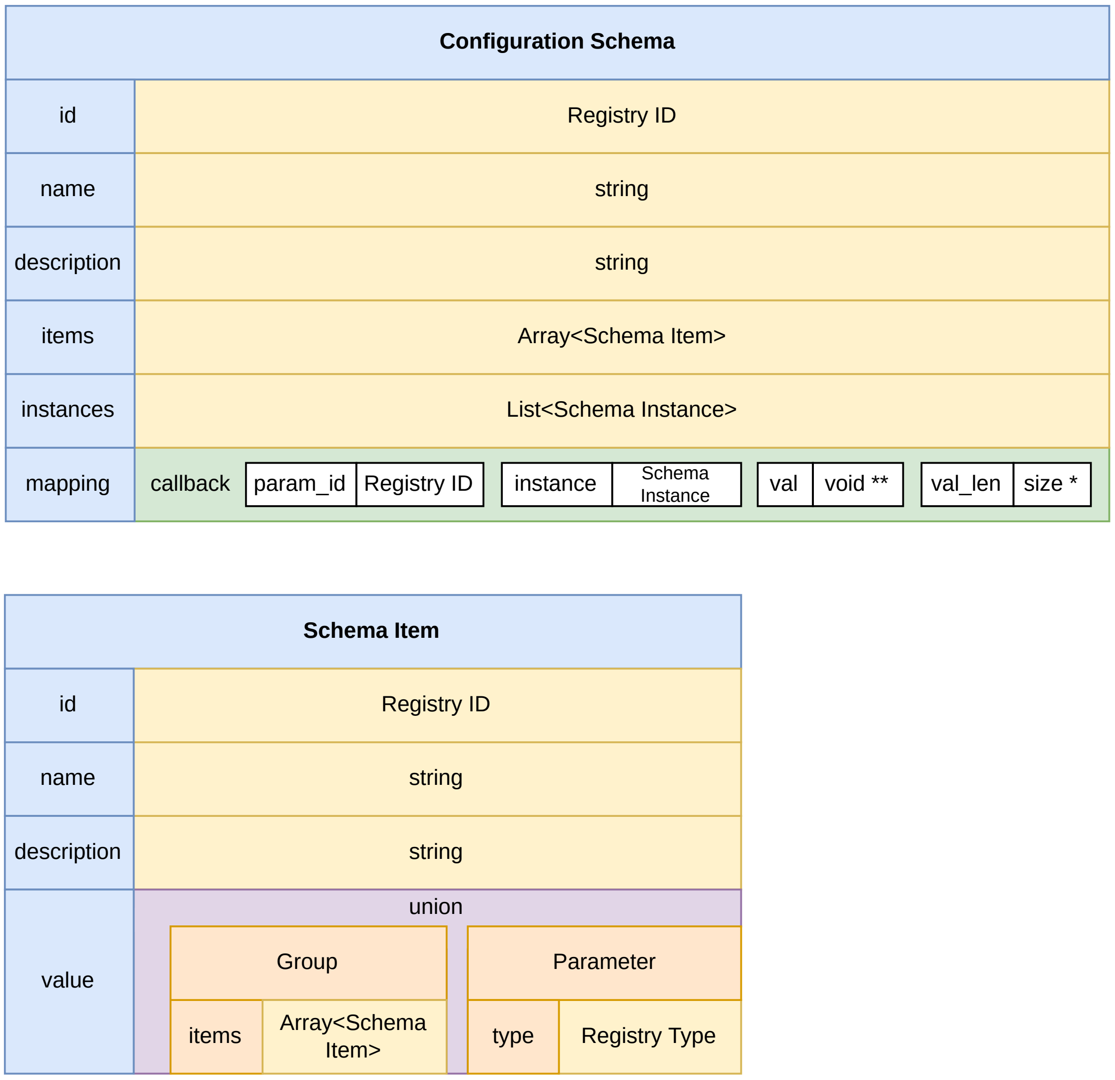


Figure 6.1: RIOT OS Registry CS implementation data structure.

Figure 6.2 shows the data structure of the “Registry Type” type. A Registry Type is an enum that can be either of type “none”, if the type is not known. This is usually used to show, that something went wrong or as a placeholder for as long as a type is not yet known. It can also have the type “opaque”. This is used to allow the RIOT OS Registry to support every other type of data that does not fit into one of the other types. It internally has the void pointer type and a specified size. Additionally a string of a fixed size, a boolean, a uint8, a uint16, a uint32, a uint64, a int8, a int16, a int32, a int64, a float32 and a float64 are supported as a Registry Type.

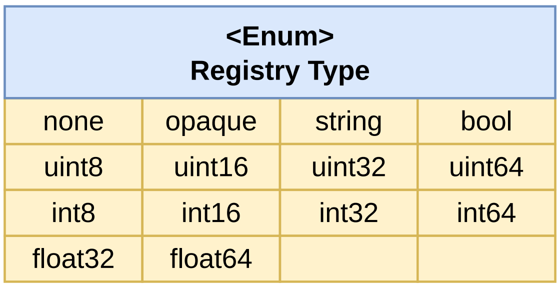


Figure 6.2: RIOT OS Registry configuration parameter type implementation enum.

Figure 6.3 shows the data structure of a SI. The “name” field allows it SI to have a human readable soft identifier. The “data” field contains its configuration parameter values in whatever format the CS implementation decides on. The “commit_cb” callback function is called whenever a configuration parameter’s new value of this SI gets “committed” (see paragraph 5.2.4). This way the module or driver that holds this SI can apply the configuration parameter changes inside the “commit_cb” function.

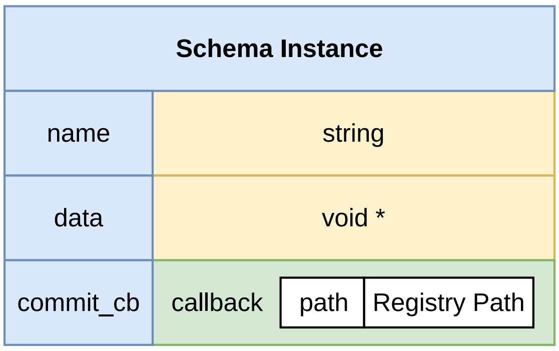


Figure 6.3: RIOT OS Registry SI implementation data structure.

Listing 6.1 shows the header file of the “registry_schemas” module with an example RGB LED CS. In line 4 the “registry_schema_init” function is defined, which registers all enabled “sys”-CN CSs at the RIOT OS Registry. From line 7 - 10 is an enum that defines which CS gets which ID. This enum prevents that two CSs use the same ID. In line 13 the CS variable is created which will be implemented in the corresponding C file. Line 16 - 21 is the struct definition of a SI. For a RGB LED CS a red, green and blue uint8 value is enough. In line 24 - 28 the configuration parameter IDs are set using an enum again. These will be used in the C file, when implementing the CS.

```
 1  #include "registry.h"
 2
 3  /* initialize schemas */
 4  void registry_schemas_init(void);
 5
 6  /* schema IDs */
 7  typedef enum {
 8      REGISTRY_SCHEMA_RGB_LED      = 1,
 9      /* SOME_OTHER_SCHEMA         = 2, ... */
10  } registry_schema_id_t;
11
12  /* RGB LED schema */
13  extern registry_schema_t registry_schema_rgb_led;
14
15  /* RGB LED instance */
16  typedef struct {
17      clist_node_t node;
18      uint8_t red;
19      uint8_t green;
20      uint8_t blue;
21  } registry_schema_rgb_led_t;
22
23  /* RGB LED configuration parameter IDs */
24  typedef enum {
25      REGISTRY_SCHEMA_RGB_LED_RED,
26      REGISTRY_SCHEMA_RGB_LED_GREEN,
27      REGISTRY_SCHEMA_RGB_LED_BLUE,
28  } registry_schema_rgb_led_indices_t;
```

Listing 6.1: Example CS implementation: registry_schemas.h

Listing 6.2 shows the source code of the “registry_schema_init.c” file. In this C file, the “registry_schemas_init” function is implemented. In line 6 it is checked if the “CON-

FIG_REGISTRY_ENABLE_SCHEMA_RGB_LED" flag to enable the RGB LED CS is set. If the flag is set, then in line 7 - 10 the RGB LED CS gets registered in the RIOT OS Registry. As the RGB LED schema is supposed to be a "sys"-CN CS, the CN is set to 0 using the "REGISTRY_ROOT_GROUP_SYS" enum value. In line 9 the in Listing 6.1 defined CS variable is passed as an argument.

```
 1  #include "registry.h"
 2  #include "registry_schemas.h"
 3
 4  void registry_schemas_init(void)
 5  {
 6      if (IS_ACTIVE(CONFIG_REGISTRY_ENABLE_SCHEMA_RGB_LED)) {
 7          registry_register_schema(
 8              REGISTRY_ROOT_GROUP_SYS,
 9              &registry_schema_rgb_led,
10          );
11      }
12  }
```

Listing 6.2: Example CS implementation: registry_schemas_init.c

Listing 6.3 shows the source code of the "registry_schema_rgb_led.c" file. It contains an example RGB LED CS implementation. To create a CS, the "REGISTRY_SCHEMA" macro can be used as seen in line 3. This macro takes the in Listing 6.1 defined CS variable, the CS ID, 2 strings and a "mapping" function as its arguments. Additionally as can be seen from line 9 - 19, it can take infinitely more configuration parameter macros ("REGISTRY_PARAMETER_*") or CG macros ("REGISTRY_GROUP") as its arguments. The latter not being present in this example. With these values, the macros generate a struct of the structure explained in Figure 6.1. In line 6 as an example, the first string will be the value of the name field and the second string will be the value of the description filed of the CS struct. Line 23 to line 47 show the "mapping" function implementation. In line 29 this function casts the data field of the SI to the correct struct of this CS's implementation. From line 31 to 46, depending on which configuration parameter ID is provided, the output pointer of a pointer ("val") gets set to the pointer of the SI's value. An example of this can be seen in line 33. The second output pointer's value ("val_len") gets set to the size of the configuration parameter's value. An example of this can be seen in line 34.

```
 1  #include "registry_schemas.h"
```

```

REGISTRY_SCHEMA(
    registry_schema_rgb_led,
    REGISTRY_SCHEMA_RGB_LED,
    "rgb", "Representation of an rgb color.",
    mapping,

    REGISTRY_PARAMETER_UINT8(
        REGISTRY_SCHEMA_RGB_LED_RED,
        "red", "Intensity of the red color of the rgb lamp.")

    REGISTRY_PARAMETER_UINT8(
        REGISTRY_SCHEMA_RGB_LED_GREEN,
        "green", "Intensity of the green color of the rgb lamp.")

    REGISTRY_PARAMETER_UINT8(
        REGISTRY_SCHEMA_RGB_LED_BLUE,
        "blue", "Intensity of the blue color of the rgb lamp.")

    );

static void mapping(
    const registry_id_t param_id,
    const registry_instance_t *instance,
    void **val,
    size_t *val_len,
) {
    registry_schema_rgb_led_t *_instance =
       (registry_schema_rgb_led_t *)instance->data;

    switch (param_id) {
    case REGISTRY_SCHEMA_RGB_LED_RED:
        *val = &_instance->red;
        *val_len = sizeof(_instance->red);
        break;

    case REGISTRY_SCHEMA_RGB_LED_GREEN:
        *val = &_instance->green;
        *val_len = sizeof(_instance->green);
        break;

    case REGISTRY_SCHEMA_RGB_LED_BLUE:
        *val = &_instance->blue;
        *val_len = sizeof(_instance->blue);
        break;
    }
}
```

Listing 6.3: Example CS implementation: registry_schema_rgb_led.c

Listing 6.4 shows an example application that uses the RGB LED CS. From line 5 - 25 the SI callback function is implemented. In this implementation this function only prints the CN, CS and SI of the committed CP, if the CP parameter "path" provides these values. From line 28 - 32 a variable that initializes an RGB LED SI struct, giving default values to the red, green and blue fields. From line 35 - 39 a variable is defined, initializing a RIOT OS Registry SI struct. This struct takes the callback function of line 5 and the RGB LED SI struct as its value. In line 44 the registry gets initialized by calling the "registry_init" function. Then in line 47 the CSs get initialized by calling the "registry_schemas_init" function. And finally from line 50 - 54 the RIOT OS Registry SI struct that got initialized in line 35, is registered in the RIOT OS Registry using the "registry_register_schema_instance" function, providing the "sys" CN and the id of the CS as additional arguments.

```
 1  #include "registry.h"
 2  #include "registry_schemas.h"
 3
 4  /* schema instance commit callback */
 5  int rgb_led_instance_0_commit_cb(
 6      const registry_path_t path,
 7      const void *context,
 8  ) {
 9      (void)context;
10
11      /* print CN, CS ID and SI ID if available */
12      printf("RGB instance commit_cb was executed: %d",
           *path.namespace_id);
13
14      if (path.schema_id) {
15          printf("/%d", *path.schema_id);
16      }
```

```

    if (path.instance_id) {
        printf("/%d", *path.instance_id);
    }

    printf("\n");

    return 0;
}

/* schema instance data struct */
registry_schema_rgb_led_t rgb_led_instance_0_data = {
    .red = 0,
    .green = 255,
    .blue = 70,
};

/* schema instance */
registry_instance_t rgb_led_instance_0 = {
    .name = "rgb-0",
    .data = &rgb_led_instance_0_data,
    .commit_cb = &rgb_led_instance_0_commit_cb,
};

int main(void)
{
    /* init registry */
    registry_init();

    /* init schemas */
    registry_schemas_init();

    /* register schema instance */
    registry_register_schema_instance(
        REGISTRY_NAMESPACE_SYS,
        registry_schema_rgb_led.id,
        &rgb_led_instance_0,
    );

    return 0;
```

```
}
```

Listing 6.4: Example CS implementation: main.c

6.2 Storage Facility (SF)

This section shows the implementation details of the in subsection 5.2.2 specified SFs. The RIOT OS Registry comes with a few officially supported SFs implemented in the "registry_storage_facilities" module. By default every SF, that is implemented inside that module is disabled, because those implementations are depending on CFLAGS to be set. The structure of those CLFAGS is the following: "DCONFIG_REGISTRY_ENABLE_STORAGE_FACILITY_{storage_facility_name}"", where "{storage_facility_name}" must be replaced with the name of an existing SF such as "VFS".

Figure 6.4 shows the structure of a SF. It consists of four callback functions. A "load" callback function that both take a "Storage Facility Instance", a CP and another callback as its arguments. Inside the load function the SF searches for configuration parameter values of the given CP. Therefor it might need for example a filesystem mount. This is specified in the data field of the "Storage Facility Instance" as a void pointer. The callback argument of the load callback takes a CP and a "Registry Value" as an argument. This callback gets executed on every configuration parameter that is found and matches the provided CP. The RIOT OS Registry sets this callback to the "registry_value_set" function. Besides the "load" callback function, the SF struct also has a "save_start" and a "safe_end" callback function. Both take the "Storage Facility Instance" as an argument. The "save_start" callback function gets called before the "save" callback function gets executed and is used to initialize the storage device, for example to mount a filesystem before multiple save operations are executed. This way unnecessary overhead of continuously mounting and unmounting filesystems can be avoided. The "save_end" callback function gets called after the "save" function finished its execution and is used to teardown the storage device, for example unmount a filesystem, after all "save" operations are completed. The "save" callback function takes a "Storage Facility Instance", a CP and a "Registry Value" as its arguments. It saves the provided configuration parameter value (Registry Value) on the storage, in a by the SF defined structure, so that the "load" function can find it by its CP.

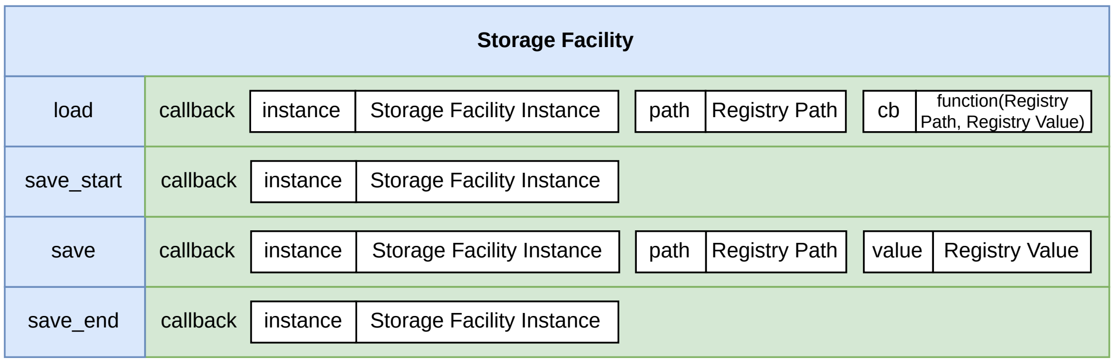


Figure 6.4: RIOT OS Registry SF implementation data structure.

Figure 6.5 shows the structure of a SF Instance. It holds a pointer to a SF implementation and provides data such as a filesystem mount as a void pointer.

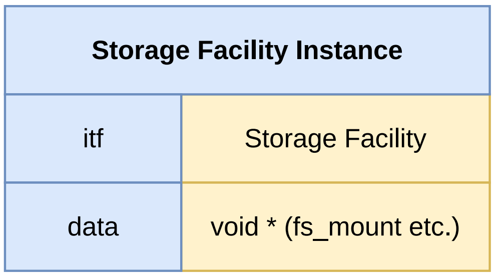


Figure 6.5: RIOT OS Registry SF Instance implementation data structure.

Figure 6.6 shows the structure of a configuration parameter called “Registry Value”. It has a “type” field which specifies the primitive type of the configuration parameter’s value. Its value is then stored in the “buf” field, which has a void pointer as its type. To pass the buffer around the program safely, it is also important to know its size, so there is a third field with the name “buf_len”, which holds the size of the value, stored in the “buf” field.

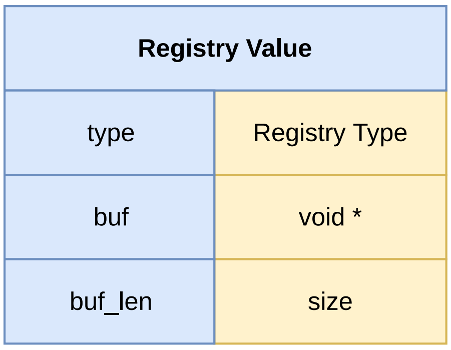


Figure 6.6: RIOT OS Registry configuration parameter value implementation structure.

Listing 6.5 shows the header file of the "registry_storage_facilities" module. It contains an example "VFS" SF variable definition called "registry_storage_facility_vfs".

```c
#include "registry.h"

/* vfs storage facility instance */
extern registry_storage_facility_t registry_storage_facility_vfs;
```

Listing 6.5: Example SF implementation: registry_storage_facilities.h

Listing 6.6 shows the implementation of the VFS SF. Because the full VFS SF implementation is around 400 lines of code, the provided SF source code in this thesis is reduced. The full source code is on the enclosed CD and on GitHub [2]. In line 23 - 26 the SF variable is initialized and the load and save callback functions are taken as arguments. The "save" function of the VFS SF creates a folder for every CN if it does not exist yet. It also creates a folder for every CS ID inside its CN folder. The same goes for the SI ID, and the CG IDs. For the configuration parameters it does not create a folder, but create a file that also uses its ID as its filename. Inside the file the configuration parameters value is written as binary. The "load" function of the VFS SF scans the storage for folders and files that match the provided CP. If those files match the CP, it asks the RIOT OS Registry for the metadata of the configuration parameter by calling the "registry_get_value" function. This way it can find out the correct size of the configuration parameters value and reads it from the storage. This value is then returned using the provided callback function.

[2]https://github.com/LasseRosenow/riot-runtime-config/blob/main/external_modules/registry_storage_facilities/storage_facility_vfs.c

```c
#include "registry_storage_facilities.h"

/* load data from storage */
static int load(
    const registry_storage_facility_instance_t *instance,
    const registry_path_t path,
    const load_cb_t cb,
    const void *cb_arg,
) {
    /* Loop through storage and call "load_cb_t" on each configuration parameter that is compatible to the specified path. */
}

/* save data to storage */
static int save(
    const registry_storage_facility_instance_t *instance,
    const registry_path_t path,
    const registry_value_t value
) {
    /* Open the file under the specified path and write the new value inside. */
}

/* storage facility */
registry_storage_facility_t registry_storage_facility_vfs = {
    .load = load,
    .save = save,
};
```

Listing 6.6: Example SF implementation: registry_storage_facility_vfs.c

Listing 6.7 shows the source code of an example application using the VFS SF. From line 6 - 14 the filesystem mount is being configured. Then in line 17 - 26 the VFS SF Instance is created, which takes the filesystem mount and the SF ("registry_storage_-facility_vfs") as its field's values. Then from line 28 - 38 the main function of the program is implemented. First in line 31 the RIOT OS Registry is initialized by calling the "registry_init" function. Then in line 34 the source SF is registered in the RIOT OS Registry by calling the "registry_register_storage_facility_src" function. And finally in line 35 the destination SF is registered in the RIOT OS Registry by calling the "registry_-register_storage_facility_dst" function.

```
 1  #include "registry.h"
 2  #include "registry_storage_facilities.h"
 3  #include "fs/littlefs2_fs.h"
 4
 5  /* initialize vfs mount */
 6  static littlefs2_desc_t fs_desc = {
 7      .lock = MUTEX_INIT,
 8  };
 9
10  static vfs_mount_t vfs_mount = {
11      .fs = &FS_DRIVER,
12      .mount_point = "/sda",
13      .private_data = &fs_desc,
14  };
15
16  /* initialize a storage facility source instance */
17  registry_storage_facility_instance_t vfs_instance_src = {
18      .itf = &registry_storage_facility_vfs,
19      .data = &vfs_mount,
20  };
21
22  /* initialize a storage facility destination instance */
23  registry_storage_facility_instance_t vfs_instance = {
24      .itf = &registry_storage_facility_vfs,
25      .data = &vfs_mount,
26  };
27
28  int main(void)
29  {
30      /* init registry */
31      registry_init();
32
33      /* register storage_facility source and destination instances */
34      registry_register_storage_facility_src(&vfs_instance_src);
35      registry_register_storage_facility_dst(&vfs_instance_dst);
36
37      return 0;
38  }
```

Listing 6.7: Example SF implementation: main.c

6.3 Configuration Path (CP)

This section shows the implementation details of the in subsection 5.2.3 specified CP.

Figure 6.7 shows the structure of the CP on the left-hand side. It contains a CN field, that is implemented as an enum containing a “sys”=0 and a “app”=1 value as possible CNs. The CP also has a CS ID field, which is of the type “Registry ID”. The Registry ID type internally is a uint32 type. The CP also has a field that holds the SI ID, which is also of the Registry ID type. And at last the CP has a “path” field, which is an array of “Registry IDs”. The “path” field contains the CG and configuration parameter IDs.

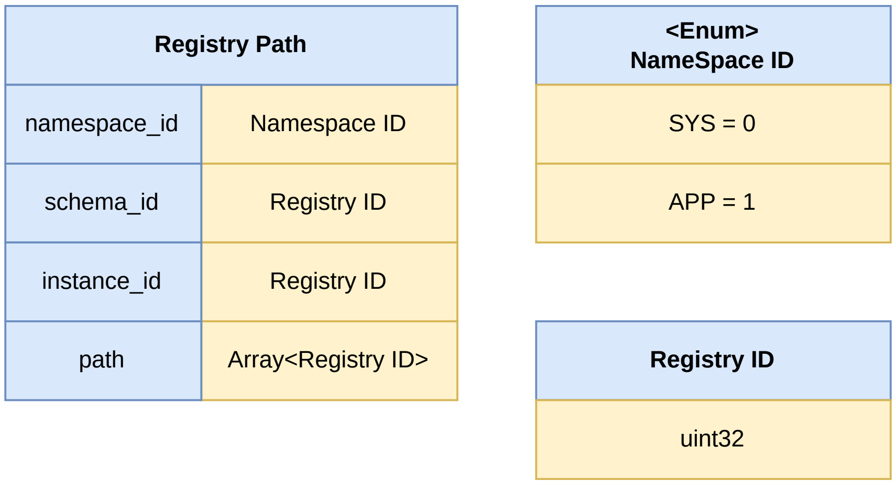


Figure 6.7: RIOT OS Registry CP implementation data structure.

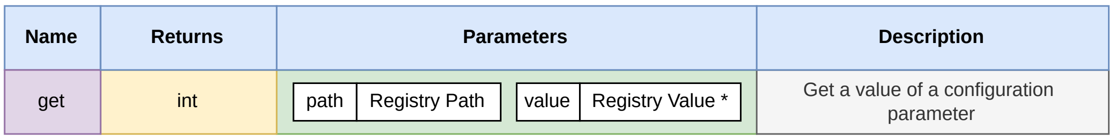
6.4 API

This section shows the implementation details of the in subsection 5.2.4 specified API.

6.4.1 Basic API

Get

Figure 6.8 shows the implementation of how to get a configuration parameter's value as specified in section 5.2. It shows a function called "get" that takes a CP and a "Registry Value" pointer as an argument. The "Registry Value" pointer is used to return the configuration parameter's value.



Name	Returns	Parameters	Description
get	int	path: Registry Path; value: Registry Value *	Get a value of a configuration parameter

Figure 6.8: RIOT OS Registry API: get.

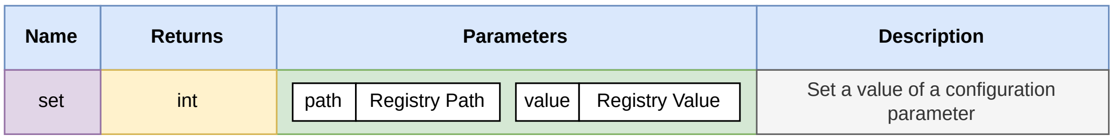
Listing 6.8 shows the source code of this function.

```c
int registry_get_value(
    const registry_path_t path,
    registry_value_t *value
);
```

Listing 6.8: Get configuration parameter values: C-function.

Set

Figure 6.9 shows the implementation of how to set a configuration parameter's value to a new value as specified in section 5.2. It shows a function called "set" that takes a CP and a "Registry Value" as an argument.



Name	Returns	Parameters	Description
set	int	path: Registry Path; value: Registry Value	Set a value of a configuration parameter

Figure 6.9: RIOT OS Registry API: set.

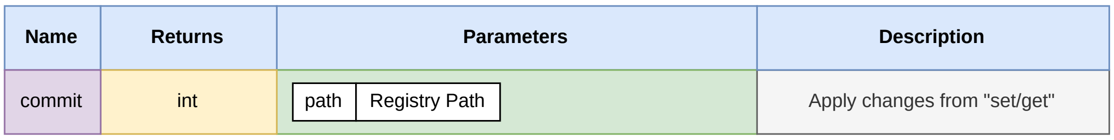
Listing 6.9 shows the source code of this function.

```c
int registry_set_value(
    const registry_path_t path,
    const registry_value_t val
);
```

Listing 6.9: Set new configuration parameter values: C-function.

Commit

Figure 6.10 shows the implementation of how to commit configuration parameters as specified in section 5.2. It shows a function called “commit” that takes a CP as an argument.



Name	Returns	Parameters	Description
commit	int	path | Registry Path	Apply changes from "set/get"

Figure 6.10: RIOT OS Registry API: commit.

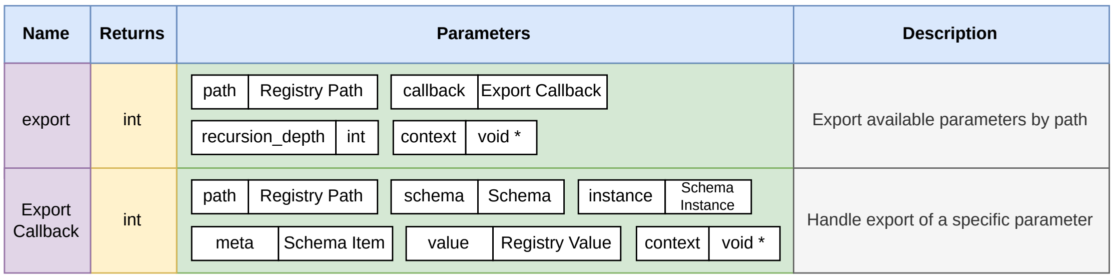
Listing 6.10 shows the source code of this function.

```c
int registry_commit(const registry_path_t path);
```

Listing 6.10: Commit configuration parameters: C-function.

Export

Figure 6.11 shows the implementation of how to export configuration parameters as specified in section 5.2. It shows a function called “export” that takes a CP, a callback function, a integer called “recursion_depth” and a void pointer as for context data as an argument. The “recursion_depth” parameter of this function specifies how deep the export function is allowed to recursively search for configuration parameters and export them. If the recursion_depth is set to 0, then all configuration parameters, that are within the specified CP are exported. If the recursion_depth is set to 1, then only a configuration parameter gets exported, if it exactly matches the given CP. If the recursion_depth is set to a higher number than 1, the export function will export all configuration parameters, that are within the given CP and are not nested more steps deeper than the given CP goes plus the specified number. The callback function takes a CP, a CS, a SI, a "Schema Item" and a "Registry Value" and a void pointer for context data as its arguments.



Name	Returns	Parameters	Description
export	int	path: Registry Path; callback: Export Callback; recursion_depth: int; context: void *	Export available parameters by path
Export Callback	int	path: Registry Path; schema: Schema; instance: Schema Instance; meta: Schema Item; value: Registry Value; context: void *	Handle export of a specific parameter

Figure 6.11: RIOT OS Registry API: export.

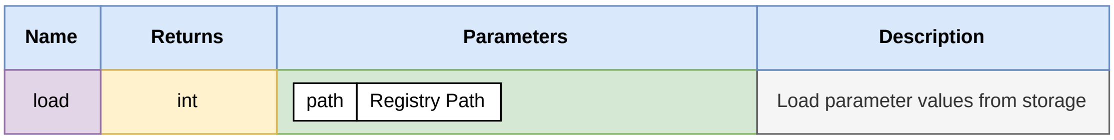
Listing 6.11 shows the source code of this function.

```c
int registry_export(
    int (*export_func)(
        const registry_path_t path,
        const registry_schema_t *schema,
        const registry_instance_t *instance,
        const registry_schema_item_t *meta,
        const registry_value_t *value,
        const void *context
    ),
    const registry_path_t path,
    const int recursion_depth,
    const void *context,
);
```

Listing 6.11: Export configuration parameters: C-function.

Load

Figure 6.12 shows the implementation of how to load configuration parameter values from a non-volatile storage device as specified in section 5.2. It shows a function called "load" that takes a CP as an argument.



Name	Returns	Parameters	Description
load	int	path | Registry Path	Load parameter values from storage

Figure 6.12: RIOT OS Registry API: load.

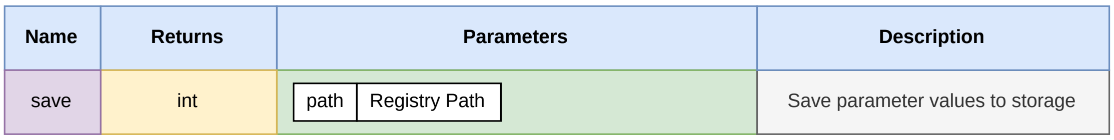
Listing 6.12 shows the source code of this function.

```c
int registry_load(const registry_path_t path);
```

Listing 6.12: Load configuration parameter values: C-function.

Save

Figure 6.13 shows the implementation of how to save configuration parameter values to a non-volatile storage device as specified in section 5.2. It shows a function called “save” that takes a CP as an argument.



Name	Returns	Parameters	Description
save	int	path | Registry Path	Save parameter values to storage

Figure 6.13: RIOT OS Registry API: save.

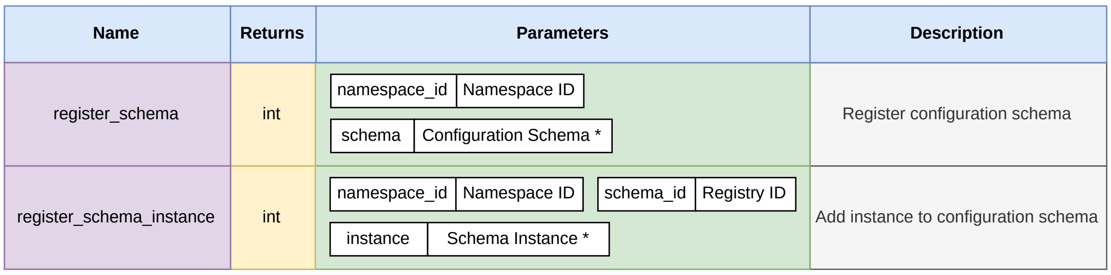
Listing 6.13 shows the source code of this function.

```c
int registry_save(const registry_path_t path);
```

Listing 6.13: Save configuration parameter values: C-function.

6.4.2 Schema Setup API

Figure 6.14 shows the implementation of how to register a CS or a IoT in the RIOT OS Registry as specified in section 5.2. It shows a function called “register_schema” that takes a CN as an argument. This function then adds the CS to a internal linked list. Figure 6.14 also shows a function called “register_schema_instance”, which takes a CN, a CS ID and a pointer to a SI as its argument. Internally the RIOT OS Registry then adds the SI to the linked list of SIs, stored in the CS that matches the given CS ID.



Name	Returns	Parameters	Description
register_schema	int	namespace_id: Namespace ID; schema: Configuration Schema *	Register configuration schema
register_schema_instance	int	namespace_id: Namespace ID; schema_id: Registry ID; instance: Schema Instance *	Add instance to configuration schema

Figure 6.14: RIOT OS Registry API: setup CS.

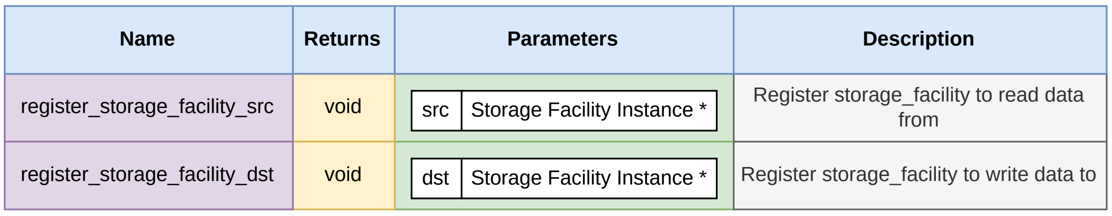
Listing 6.14 shows the source code of these functions.

```
void registry_schemas_init(void);

int registry_register_schema(
    const registry_namespace_id_t namespace_id,
    const registry_schema_t *schema
);

int registry_register_schema_instance(
    const registry_namespace_id_t namespace_id,
    const registry_id_t schema_id,
    const registry_instance_t *instance
);
```

Listing 6.14: Schema Setup API.

6.4.3 Storage Facility Setup API

Figure 6.15 shows the implementation of how to register a SF in the RIOT OS Registry as specified in section 5.2. It shows a function called "register_storage_facility_src" that takes a pointer to a SF Instance as an argument. This function then adds the SF Instance to a internal linked list of SF sources. Figure 6.15 also shows a function called "register_storage_facility_dst" that takes a pointer to a SF Instance as an argument. This function then sets the SF Instance as the internal SF. This SF is not added to a internal linked list, because there can only be one SF to write to.



Name	Returns	Parameters	Description
register_storage_facility_src	void	src Storage Facility Instance *	Register storage_facility to read data from
register_storage_facility_dst	void	dst Storage Facility Instance *	Register storage_facility to write data to

Figure 6.15: RIOT OS Registry API: Setup SF.

Listing 6.15 shows the source code of these functions.

```c
void registry_register_storage_facility_src(
    const registry_storage_facility_instance_t *src
);

void registry_register_storage_facility_dst(
    const registry_storage_facility_instance_t *dst
);
```

Listing 6.15: Storage Facility Setup API.

7 Testing of the Implementation's Correctness

The "registry_tests" module provides unit tests that can be run by including the "registry_tests.h" header file and calling the "registry_tests_api_run" function.

7.1 Test Setup

To be able to test if all by the RIOT OS Registry supported data types are supported, the "registry_tests" module uses the "registry_schema_full_example" of the "registry_schemas" module. This CS contains fields for every supported data type of the RIOT OS Registry (see section 3.5 and Figure 6.2). Additionally the module creates and registers a custom CS called "registry_schema_groups_test", which consists of 4 CGs, that are children of each other, creating a maximum CP of up to 5 segments excluding the CN, CS and SI. This is necessary to test if the CGs are implemented correctly.

7.2 Testing the "registry_get" and "registry_set" Functions

To test the correctness of the "registry_get" and "registry_set" functions of the RIOT OS Registry, first we call the "registry_set" function for all fields of the "registry_schema_full_example" CS and also for all fields of the "registry_schema_groups_test" CS to test every data type that is supported by the RIOT OS Registry and also if CGs work. Then we call the "registry_get" function for all previously set parameters and compare the returning values to the original values. If these values match, the test is successful. This testing sequence is executed for the minimum and maximum value for each configuration parameter.

7.3 Testing the "registry_commit" Function

To test the correctness of the "registry_commit" function of the RIOT OS Registry, first we create a global bool and initialize it to "false". Then we call the "registry_commit" function providing a CP as an argument that points to a configuration parameter of a SI, which in its callback function checks, if the provided values are matching to what is expected. It then changes the value of the global bool to "true" if the values are correct and to "false" if they are not. If the value is set to "true", the test is successful.

Ideally this test is not only implemented for calling a concrete configuration parameter of a SI, but also for calling incomplete CPs, such as a CP that only points to the CS or one that only points to the SI. In both cases the configuration parameter must be committed because its parent got committed. These tests are not yet implemented.

7.4 Testing the "registry_export" Function

To test the correctness of the "registry_export" function of the RIOT OS Registry, first we create a global bool and initialize it to "false". Then we call the "registry_export" function providing a CP that points to a configuration parameter and a custom callback function as its arguments. When the callback function gets called, it checks if the provided values are matching to what is expected. It then changes the value of the global bool to "true" if the values are correct and to "false" if they are not. If the value is set to "true", the test is successful.

Ideally this test is not only implemented for calling a concrete configuration parameter of a SI, but also for calling incomplete CPs, such as a CP that only points to the CS or one that only points to the SI. In these cases it depends on the "recursion" argument, how deep the "registry_export" function searches inside the CS for configuration parameters to export. These tests are not yet implemented.

7.5 Testing the "registry_save" and "registry_load" Function

To test the "registry_save" and "registry_load" functions, every test that is done in 7.2 gets executed again, but this time after each sequence we call the "registry_save" function to write the values to the storage. Then all previously set configuration parameters get changed to a different value using the "registry_set" function. And finally we call the "registry_load" function to read the values written to storage and loading them into the RIOT OS Registry again. If these values now match with the values that were initially set by the "registry_set" function, then the test is successful.

8 Evaluation of the implementation's overhead

8.1 RAM

8.1.1 Heap

The RIOT OS Registry does not do any dynamic heap allocations, so evaluating the heap overhead is not necessary.

8.1.2 Stack

In this section the stack consumption of the RIOT OS Registry API is measured and discussed. More specifically the functions "registry_get_value", registry_set_value, "registry_commit", "registry_export", "registry_save" and "registry_load" are measured.

Method

To measure the stack usage of a function in RIOT OS is possible, because the stack implementation internally marks parts of the stack that have been "touched". This way the highest ever used stack count can be printed out, but this also has the downside that before starting the measurement, the currently highest measured stack count must be equal to the current stack count. This can be achieved by creating a new thread, which start stack count is equal to the highest stack count until that moment.

This leads to the following stack consumption measurement strategy:

1. Create a new thread.

2. Get the current stack count (highest).
3. Save the current stack count to a variable.
4. Call the function which stack consumption needs to be measured.
5. Get the highest stack count.
6. Subtract the old stack count from the new stack count.
7. Print out the result to the terminal.

Measurements

Table 8.1 and Figure 8.1 show the result of this measurement in bytes. The first row of Table 8.1 tells the length of the CP, all following rows show the stack consumption of each function per CP length. The result shows that almost all functions don't increase their overall stack consumption at all if the CP length changes. One exception is "registry_-export", which has a increase of 16 bytes (+1.5%) from a CP with the length of 1 to a CP with the length of 2 and it also has another increase of 16 bytes from a CP with the length of 4 to a CP with the length of 5. The other exception is the function "registry_save", which oscillates between a stack consumption of up to 3,900 and down to 3,340.

Function \ CP length	1	2	3	4	5
registry_get_value	276	276	276	276	276
registry_set_value	308	308	308	308	308
registry_commit	464	464	464	464	464
registry_export	1,092	1,108	1,108	1,108	1,124
registry_save	3,884	3,340	3,900	3,340	3,356
registry_load	2,636	2,636	2,636	2,636	2,636

Table 8.1: Stack consumption of RIOT OS Registry API functions in bytes.

Figure 8.1 shows the stack consumption of each main RIOT OS Registry function on the y axis in bytes. The x axis shows the length of the CP that was passed to each function.

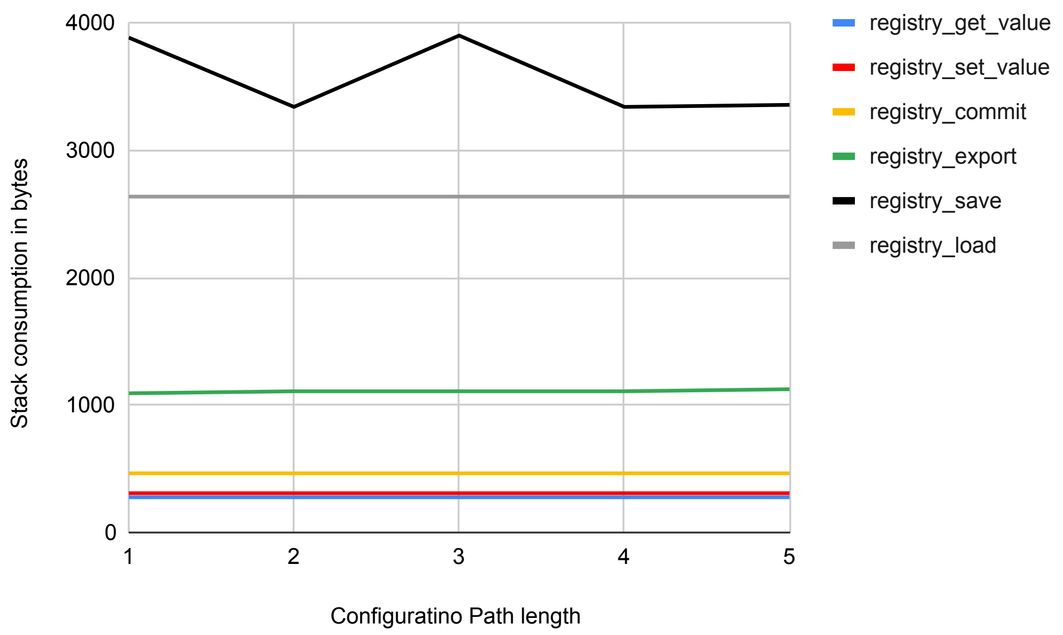


Figure 8.1: RIOT OS Registry stack consumption in bytes per API function on the CP lengths 0 - 5.

Discussion

Overall the results of these measurements are not unexpected and possible to explain. The "set" and "get" function are implemented without recursion and should not consume more stack only because they need to go deeper into the CS structure. The "commit" function has a similar implementation to the "get" and "set" functions, but still consumes more stack because its slightly more complex implementation is currently distributed among multiple functions that call each other and it also needs to call the callback function of the SI to inform the driver / module about the to be applied changes (see Figure 5.7 and Figure 5.6 for comparison). The "export" function is similar implemented to the "commit" function (see Figure 5.8) and has its logic distributed among multiple functions, that call each other. The major difference, that also causes the much higher stack usage is that each of these functions take up to 7 times more arguments.

The stack consumption of the "registry_save" and "registry_load" functions however are more difficult to discuss. In this example both use a SF that internally uses the RIOT OS Virtual File System (VFS) layer, which abstracts over filesystems such as FatFs or littlefs. In this case the littlefs abstraction is used. What happens with littlefs internally is unknown, but the measurements behave reasonable in contrast to how the VFS SF is implemented. Both functions do not use recursive function calls and only rely on while loops that reuse data structures. As a consequence also these two function don not consume more stack in relation to the CP length. But this conclusion is only valid for the VFS SF. Other implementations may show different results.

In general the implementation of all the analyzed functions can be improved. Especially the distribution into multiple functions that call each other by passing on a lot of arguments is not ideal. It makes the source code less difficult to maintain, but has the consequence of consuming unnecessarily much stack. It should theoretically be possible to make the functions "get", "set", "commit" and "export" consume almost the same amount of stack with a more better implementation. Of course there would still be some differences caused by the different requirements of these functions. Such things as the callback of the "commit" function.

8.2 ROM

RIOT OS has a tool called "Cosy". It can be used as a build target and analyzes the last build binary size in different categories. To compare the ROM size of a RIOT OS application with and without the RIOT OS Registry, the application is analyzed by the given tool.

8.2.1 Full Binary Size Comparison

Method

First, the ROM size of a sample RIOT OS application is measured with all RIOT OS Registry modules disabled in the makefile. Then the RIOT OS Registry modules get enabled one by one and new measurements are taken. At the end the ROM size of all modules together is measured as well. Subtracting the initial ROM size with all RIOT OS Registry modules disabled from the measurements that enable some additional RIOT OS Registry modules, allows to see the actual ROM size cost of using the RIOT OS Registry modules.

Measurements

The measured size in bytes of the sample application without any RIOT OS modules enabled can be seen in Table 8.2. The measurements with each module separately enabled and also all at once can be seen in Table 8.3, which shows the name of the enabled modules in column one, the total measured ROM size in bytes in column two, the difference between with and without the module can be found in column three in bytes and in column 4 in percentage relative to the ROM size without any RIOT OS Registry modules enabled.

Enabled RIOT OS Registry module(s)	Size in bytes
-	1,386,602

Table 8.2: ROM size without any RIOT OS Registry modules enabled.

Enabled RIOT OS Registry module(s)	Size in bytes	Difference in bytes	Difference in %
registry	1,437,782	+51,180	+3.7
registry_schemas	1,416,356	+29,754	+2.1
registry_storage_facilities	1,419,947	+33,345	+2.4
registry_cli	1,407,312	+20,710	+1.5
All registry_* modules	1,503,139	+116,537	+8.4

Table 8.3: ROM size of RIOT OS Registry modules and their overhead in bytes and percentage.

Discussion

It should be noted that if the ROM size differences of each RIOT OS Registry module get accumulated, the result (134,989) is 18,452 bytes larger then if all RIOT OS Registry modules are enabled together. This is caused by shared dependencies that get optimized by the compiler.

Besides the RIOT OS Registry modules listed in Table 8.3, there is also the “registry_-tests” module. This module only contains tests and is not needed when using the RIOT OS Registry inside an application. That is why it is not part of Table 8.3, which means it is also not part of the “All registry_* modules” enabled row.

8.2.2 Compiled Object Sizes

Method

The "Cosy" tool also allows to show the exact size of compiled object files. This allows to get a more precise view of how much ROM is exactly used per module implementation.

Measurements

To be able directly compare with the previous measurements listed in Table 8.3, in Table 8.4 the object file sizes are not provided alone, but also accumulated per RIOT OS module. By comparing the sizes of column 2 in Table 8.4 to the calculated differences in column 3 of Table 8.3, it can be said, that each of the corresponding values have around the same size. In most cases the object files based module size is between 3 to 4.5 kilobytes smaller, except in case of the "registry" module, which is around 4.5 kilobytes larger then the measurement of Table 8.3.

Module	Module size in bytes	Object file	Object file size in bytes
registry	55,678	registry.o	39,447
		registry_conversion.o	16,231
registry_schemas	25,219	registry_schemas_init.o	5,726
		registry_schema_rgb_led.o	8,834
		registry_schema_full_example.o	10,659
registry_ storage_facilities	30,335	registry_storage_facility_vfs.o	21,189
		registry_storage_facility_heap.o	9,146
registry_cli	17,824	registry_cli.o	17,824
registry_tests	53,128	registry_tests_stack.o	32,597
		registry_tests_api.o	20,531

Table 8.4: Compiled object file size of RIOT OS Registry modules.

Figure 8.2 shows the same results as Table 8.4, but as a donut diagram to better understand the relation between the measured objects. The inner circle represents the RIOT OS Registry modules and the outer circle represents the corresponding object files that were compiled from the c-files implementing these modules.

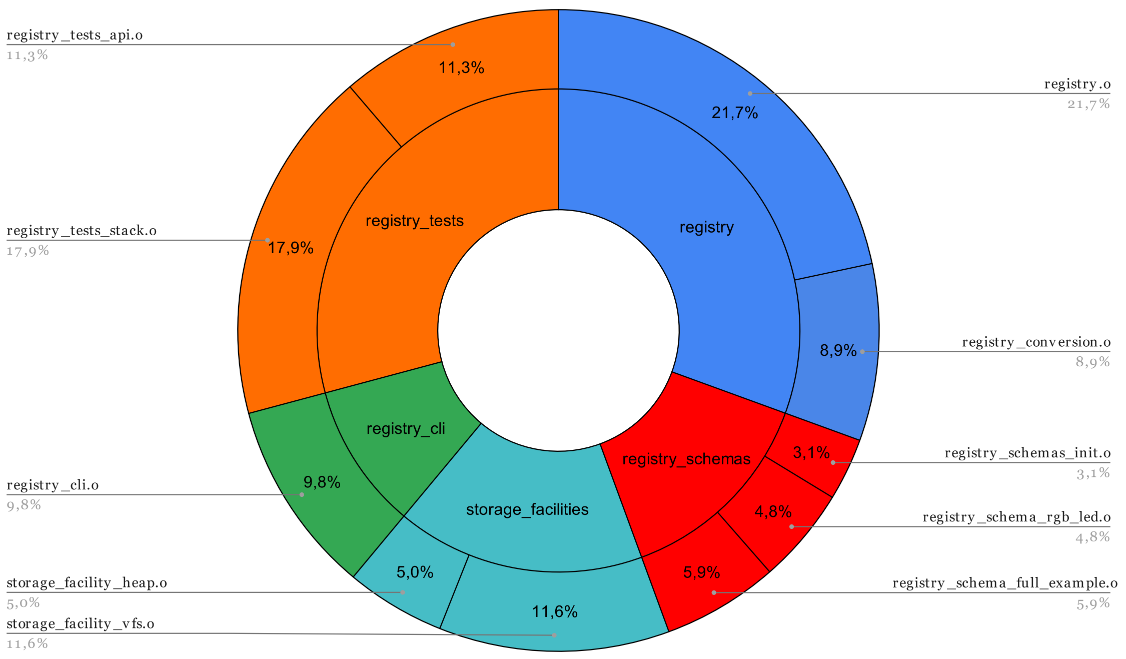


Figure 8.2: RIOT OS Registry ROM usage per module (inner circle) and per object file (outer circle).

Discussion

The more fine-grained measurements are helpful because the "registry_schemas" and "registry_storage_facilites" modules allow to enable and disable each CS and SF implementation separately. This way a lot of ROM overhead can be avoided by only enabling what is needed and these measurements show about how much can be saved.

It should also be noted that the "registry_tests" module is usually not enabled when using the RIOT OS Registry.

9 Future Work

9.1 Full Test Coverage

As mentioned in 7, the currently implemented tests don't have the highest test coverage yet. In the future the unit tests of the RIOT OS Registry should be extended to find and prevent more implementation mistakes.

9.2 Exposing Configuration Parameters Beyond Abstraction

Investigate how to expose custom config data beyond abstraction: Some sensors have very specific configuration parameters that can not all be sufficiently abstracted through one large interface. Possible solutions:

- Implement custom CS for these sensors.
- Expose those parameters through a field in every CS that can hold additional parameters and is not type-safe. Set, Get handlers and persistence must be implemented by the driver itself and not the CS in this case.

9.3 Extend Configuration Parameter Value Constraints

Investigate how to expose further value constraints of configuration parameters. Configuration parameters, in the case of numbers, usually for example don't expect the full range that the size of a primitive c-type (8, 16, 32 or 64-bit) allows. In some cases maybe even only a certain set of specifically selected values is a correct configuration value.

There should be a way to specify how to expose those constraints through the Registry API through additional metadata fields. Possible use-cases:

- Constrain the minimum and maximum value of a numeric configuration parameter.
- Constrain the minimum and maximum length of a (string) configuration parameter.
- Specifies a set of values that are allowed to be set. This gives the possibility to implicitly define an "enum" like configuration parameter.

9.4 External Configuration Manager Implementation

This thesis only specifies the architectural design of how CoAP, MQTT and LwM2M can be used to implement external Configuration Managers for the RIOT OS and only implements a CLI for external configuration management (see subsubsection 5.3.1). Those integrations still need to be implemented.

- CoAP based API (uses registry structure).
- MQTT based API (uses registry structure).
- LwM2M mapping (RIOT OS Registry => LwM2M Object Models).

9.5 Specification of Sys Configuration Schemas

As of now the RIOT OS Registry only comes with two simple example CSs in its "sys" CN, one being an LED-RGB CS to control the color of an rgb LED, and the other being a CS with the name "full_example" that contains all supported parameter types for testing. To give modules/drivers the ability to implement CSs, first those CSs must be defined. This is no simple task as each CS must be specified in a way that is compatible to as many as possible drivers of the same kind.

9.6 Integration of the RIOT OS Registry into RIOT OS Modules and Drivers.

After a CS is specified as explained in section 9.5, it has to be implemented by the compatible modules/drivers that need runtime configuration.

10 Conclusion

Our main goal of this thesis to specify a runtime configuration registry for RIOT OS has been achieved (see chapter 5). The implementation (see chapter 6) has also been successful and fulfills all the requirements specified in chapter 3.

Looking back and around to see what has already been done prior to this thesis in chapter 4 has been very helpful for us to get an overview of in what ways RCSs have been designed so far and for which use-cases. Thinking about the use-cases of already existing solutions helped in specifying use-cases and requirements that are relevant to fulfill, for the RIOT OS implementation even if the related work was not exactly what is needed for a RIOT OS RCS. This in turn helped us to assess how much the in chapter 4 assessed already existing implementations are capable of fulfilling these RIOT OS-specific requirements (see chapter 3 and chapter 4 section 4.3). We learned that none of the assessed already existing implementations fulfill all the requirements of chapter 3, but we were able to base the new RIOT OS Registry on the existing Mynewt Config implementation for RIOT OS, which helped speed up the development process by a lot. Our final design (see chapter 5) turned out much different to from where it started (Mynewt Config), but the main API functions are still roughly the same.

Writing unit tests (see chapter 7) helped not only finding a lot of bugs in the RIOT OS Registry implementation, but also helped in creating new ones while further iterating on the RIOT OS Registry's implementation. Which is why, it is of high importance to further increase the testing coverage as mentioned in chapter 9.

chapter 8 showed us that our implementation's overhead is acceptable for devices with enough memory (stack overhead up to $>$ 4 kilobyte) and it does not perform any dynamic heap allocations, but there are still many ways to further reduce its stack overhead.

For the future it will be very interesting to see how good the new RIOT OS Registry can specify CSs that abstract common configuration parameters shared by the many RIOT OS modules and drivers (see section 9.5).

Bibliography

[1] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," IETF, RFC 7252, June 2014.

[2] C. Bormann and P. Hoffman, "Concise Binary Object Representation (CBOR)," IETF, RFC 8949, December 2020.

[3] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," IETF, RFC 8259, December 2017.

[4] Apache Software Foundation, "Apache Mynewt," https://mynewt.apache.org, last accessed 07-17-2020, 2020.

[5] O. SpecWorks, "Lightweight Machine to Machine Technical Specification: Core v1.2," Open Mobile Alliance, Tech. Rep., 2020.

[6] "matter standard," https://csa-iot.org/all-solutions/matter/, [Online; accessed 07-01-2023].

[7] E. Baccelli, C. Gündogan, O. Hahm, P. Kietzmann, M. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, December 2018. [Online]. Available: http://dx.doi.org/10.1109/JIOT.2018.2815038

[8] "Riot os documentation," https://doc.riot-os.org/, [Online; accessed 06-01-2023].

[9] *GNU Make*, https://www.gnu.org/software/make/manual/make.pdf, [Online; accessed 14-01-2023].

[10] The Linux Kernel Development Community, "Kconfig Language," https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html, last accessed 28-09-2020, 2020.

[11] V. Rana, M. Santambrogio, and D. Sciuto, "Dynamic reconfigurability in embedded system design," in *2007 IEEE International Symposium on Circuits and Systems*, 2007, pp. 2734–2737.

[12] M. Lenders, P. Kietzmann, O. Hahm, H. Petersen, C. Gündogan, E. Baccelli, K. Schleiser, T. C. Schmidt, and M. Wählisch, "Connecting the World of Embedded Mobiles: The RIOT Approach to Ubiquitous Networking for the Internet of Things," Open Archive: arXiv.org, Technical Report arXiv:1801.02833, January 2018. [Online]. Available: https://arxiv.org/abs/1801.02833

[13] "Riot os drivers," https://doc.riot-os.org/group__drivers.html, [Online; accessed 08-01-2023].

[14] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," IETF, RFC 3986, January 2005.

[15] A. Banks and R. G. (Eds.), "MQTT Version 3.1.1," OASIS, OASIS Standard, October 2014. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html

[16] F. Adelantado, X. Vilajosana, P. Tuset-Peiro, B. Martinez, J. Melia-Segui, and T. Watteyne, "Understanding the Limits of LoRaWAN," *Communications Magazine*, vol. 55, no. 9, pp. 34–40, Sep. 2017.

[17] M. Veillette, P. van der Stok, A. Pelov, A. Bierman, and I. Petrov, "CoAP Management Interface (CORECONF)," IETF, Internet-Draft – work in progress 11, January 2021.

[18] M. Hussein, S. Li, and A. Radermacher, "Model-driven development of adaptive iot systems." in *MODELS (Satellite Events)*, 2017, pp. 17–23.

[19] B. Costa, P. F. Pires, and F. C. Delicato, "Modeling iot applications with sysml4iot," in *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2016, pp. 157–164.

[20] F. Alkhabbas, R. Spalazzese, and P. Davidsson, "Architecting emergent configurations in the internet of things," in *2017 IEEE International Conference on Software Architecture (ICSA)*, 2017, pp. 221–224.

[21] M. Bjorklund, "YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)," IETF, RFC 6020, October 2010.

[22] Zephyr Project, "Zephyr," https://www.zephyrproject.org, last accessed 07-17-2020, 2020.

[23] "Zephyr settings pr," https://github.com/zephyrproject-rtos/zephyr/pull/6408, [Online; accessed 26-09-2022].

[24] "Eclipse wakaama," https://github.com/eclipse/wakaama, [Online; accessed 04-01-2023].

[25] "Riot pr 10622," https://github.com/RIOT-OS/RIOT/pull/10622, [Online; accessed 24-11-2022].

[26] "Riot pr 10799," https://github.com/RIOT-OS/RIOT/pull/10799, [Online; accessed 24-11-2022].

[27] "Wikipedia," https://wikipedia.org, [Online; accessed 19-01-2023].

[28] "Lightweight m2m (lwm2m)," https://omaspecworks.org/what-is-oma-specworks/iot/lightweight-m2m-lwm2m, [Online; accessed 13-01-2023].

[29] "Sk6812," https://cdn-shop.adafruit.com/product-files/1138/SK6812+LED+datasheet+.pdf, [Online; accessed 20-01-2023].

[30] "Ucs1903," https://cdn.sparkfun.com/assets/6/d/6/c/3/UCS1903_IC-manul.pdf, [Online; accessed 20-01-2023].

[31] "Ws2812," https://cdn-shop.adafruit.com/datasheets/WS2812.pdf, [Online; accessed 20-01-2023].

Glossary

Configuration Manager In the context of this thesis a Configuration Manager is an application that allows to change configurations of configurable devices at runtime. .

LwM2M “OMA SpecWorks’ LightweightM2M is a device management protocol designed for sensor networks and the demands of a machine-to-machine (M2M) environment. With LwM2M, OMA SpecWorks has responded to demand in the market for a common standard for managing lightweight and low power devices on a variety of networks necessary to realize the potential of IoT. The LwM2M protocol, designed for remote management of M2M devices and related service enablement, features a modern architectural design based on REST, defines an extensible resource and data model and builds on an efficient secure data transfer standard called the Constrained Application Protocol (CoAP). LwM2M has been specified by a group of industry experts at the OMA SpecWorks Device Management Working Group and is based on protocol and security standards from the IETF” [28, 5] .

MQTT “MQTT is a Client Server publish/subscribe messaging transport protocol. It is light weight, open, simple, and designed so as to be easy to implement. These characteristics make it ideal for use in many situations, including constrained environments such as for communication in Machine to Machine (M2M) and Internet of Things (IoT) contexts where a small code footprint is required and/or network bandwidth is at a premium” [15] .

RIOT OS “As the Internet of Things (IoT) emerges, compact operating systems (OSs) are required on low-end devices to ease development and portability of IoT applications. RIOT is a prominent free and open source OS in this space.” [7] .

SK6812 “SK6812 is a set of smart control circuit and a light emitting circuit in one of the controlled LED source” [29] .

UCS1903 “3-Channel Constant Current LED Driver UCS1903” [30] .

WS2812 “WS2812 is a intelligent control LED light source that the control circuit and RGB chip are integrated in a package of 5050 components” [31] .

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original