Runtime Configuration

inside RIOT

Lasse Rosenow

June 21, 2022

Table of Contents

Motivation

Motivation

Context
Many applications in IoT use parameters that need to be changed at runtime.

- Authentication credentials
- Sampling rate of a measurement
- LED color

Problem
RIOT does not provide an API for runtime parameters.

$\Rightarrow$ Each application has to implement its own runtime configuration strategy.

$\Rightarrow$ Unnecessary and redundant implementation effort

Solution: RIOT-Registry

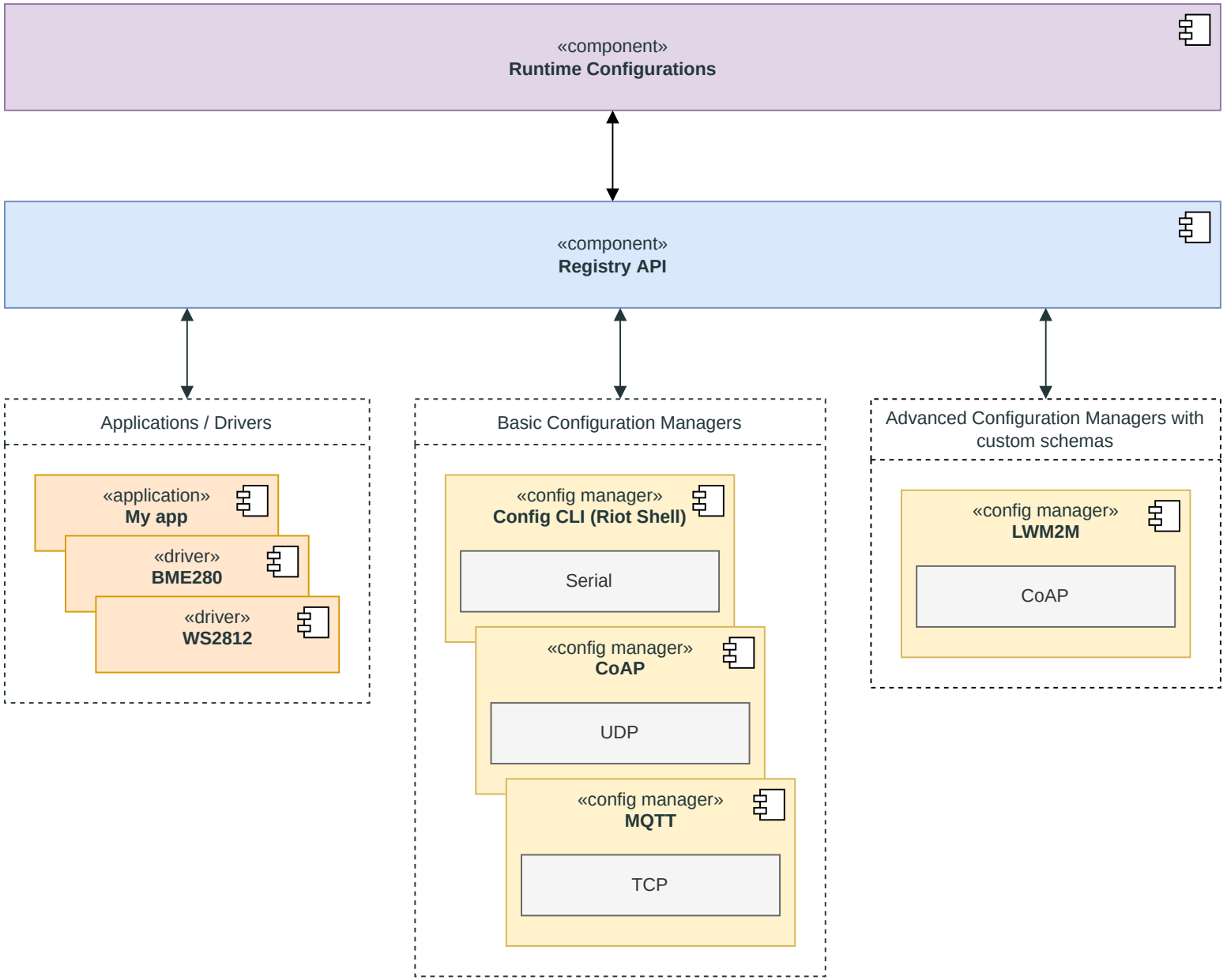
Solution: RIOT-Registry

Registry to manage configurations of RIOT modules and applications

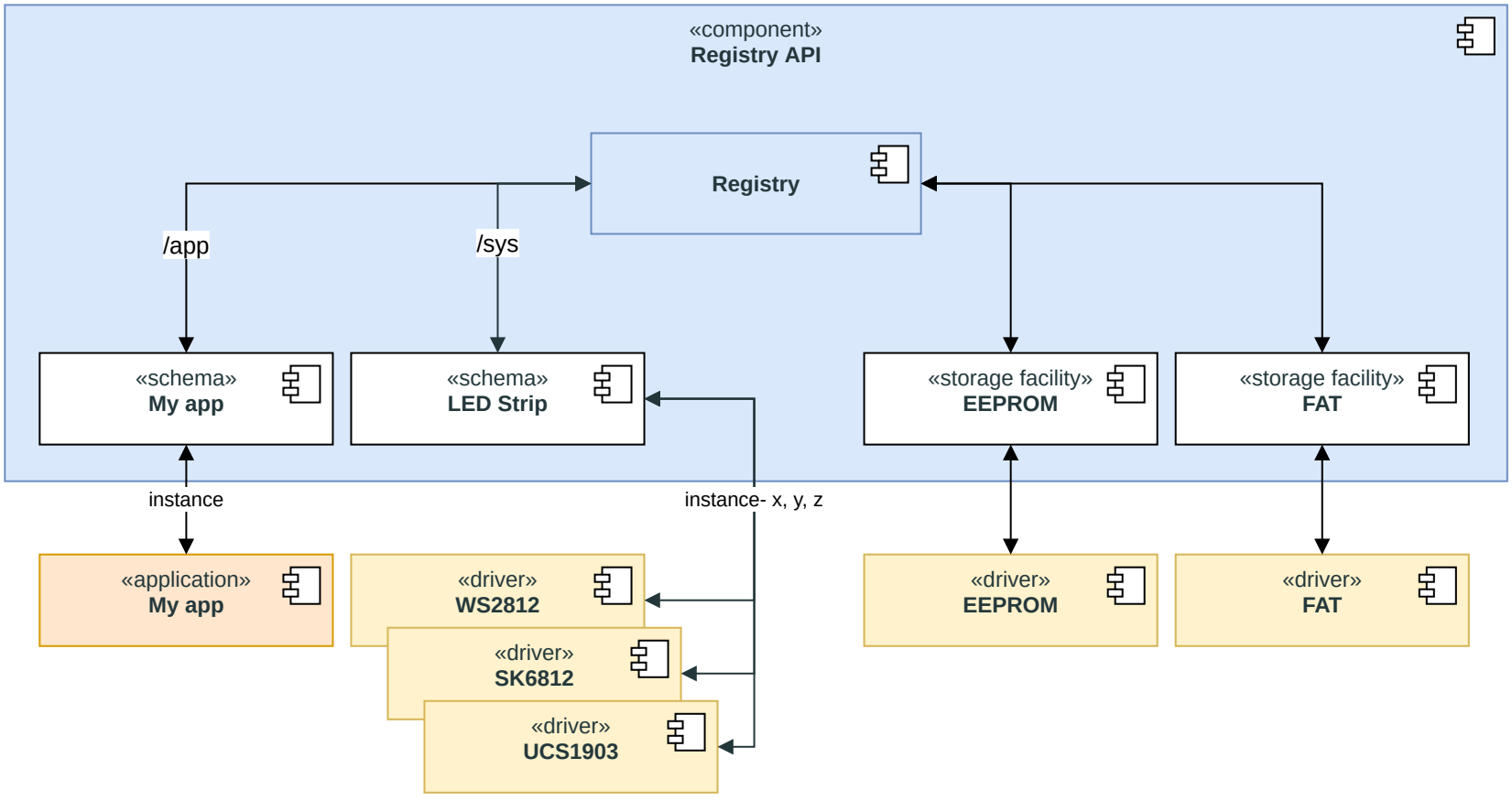
- Based on previous work by Leandro and José et al (RIOT PR 10622 and 10799)
- Contains predefined schemas for common configuration scenarios
- Configuration parameters are identified via int path: *root_group_id/schema_id/instance_id/(\d+/)*parameter_id*
- Optional:
  - Persistent storage of configuration values
  - Local access via USB/UART (CLI, etc.)
  - Remote access via (CoAP, LwM2M, MQTT, etc.)

Architecture

Architecture: Overview



Architecture: Registry Components



Architecture: Registry Path

- Unique path to a parameter or group.
- Array of int32
- *root_group_id/schema_id/instance_id/(\d+/)*parameter_id*

Registry Path	
root_group_id	Root Group ID
schema_id	int32
instance_id	int32
path	Array<int32>

<Enum> Root Group ID
SYS = 0
APP = 1

Architecture: Registry Type

- Enum containing information about the type of a configuration value

<Enum> Registry Type		
string	uint8	int8
bool	uint16	int16
float32	uint32	int32
float64	uint64	int64

Architecture: Registry Value

- Contains the value of a configuration parameter
- Value is encoded depending on the type

Registry Value	
type	Registry Type
buf	void *
buf_len	int

Architecture: Configuration Schemas

- Interface between the RIOT Registry and modules / apps
- Defines configurable parameters and their types
- Does not contain actual parameter values, but points to a list of instances
- get/set configurations of a given instance

Architecture: Configuration Schemas: Data Model

Schema						
id	int32					
name	string					
description	string					
items	Array<Schema Item>					
instances	List<Schema Instance>					
get	callback	param_id	int32	instance	Schema Instance	buf	void *
set	callback	param_id	int32	instance	Schema Instance	val	void *

Schema Instance			
name	string		
data	void *		
commit_cb	callback	path	Registry Path

Schema Item				
id	int32			
name	string			
description	string			
value	union			
	Group		Parameter	
	items	Array<Schema Item>	type	Registry Type

Architecture: Configuration Schemas: Code Example

led.c

```c
REGISTRY_SCHEMA(
    registry_schema_rgb_led,
    REGISTRY_SCHEMA_RGB_LED,
    "rgb", "Representation of an rgb color.",
    get, set,

    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 get(int param_id, registry_instance_t *instance, void *buf, int buf_len,
                void *context)
{
    (void)buf_len;
    (void)context;

    registry_schema_rgb_led_t *_instance = (registry_schema_rgb_led_t *)instance->data;

    switch (param_id) {
    case REGISTRY_SCHEMA_RGB_LED_RED:
        memcpy(buf, &_instance->red, sizeof(_instance->red));
        break;

    case REGISTRY_SCHEMA_RGB_LED_GREEN:
        memcpy(buf, &_instance->green, sizeof(_instance->green));
        break;

    case REGISTRY_SCHEMA_RGB_LED_BLUE:
        memcpy(buf, &_instance->blue, sizeof(_instance->blue));
        break;
    }
}
```

led.h

```c
extern registry_schema_t registry_schema_rgb_led;

typedef struct {
    clist_node_t node;
    uint8_t red;
    uint8_t green;
    uint8_t blue;
} registry_schema_rgb_led_t;

typedef enum {
    REGISTRY_SCHEMA_RGB_LED_RED,
    REGISTRY_SCHEMA_RGB_LED_GREEN,
    REGISTRY_SCHEMA_RGB_LED_BLUE,
} registry_schema_rgb_led_indices_t;
```

Architecture: Schema Instance: Code Example

```c
int rgb_led_instance_0_commit_cb(const registry_path_t path, void *context)
{
    (void)context;
    printf("RGB instance commit_cb was executed: %d", *path.root_group_id);
    if (path.schema_id) {
        printf("/%d", *path.schema_id);
    }
    if (path.instance_id) {
        printf("/%d", *path.instance_id);
    }
    printf("\n");
    return 0;
}

registry_schema_rgb_led_t rgb_led_instance_0_data = {
    .red = 0,
    .green = 255,
    .blue = 70,
};
registry_instance_t rgb_led_instance_0 = {
    .name = "rgb-0",
    .data = &rgb_led_instance_0_data,
    .commit_cb = &rgb_led_instance_0_commit_cb,
};
```

Architecture: Storage Facilities

- Load/store configurations from/to storage
- Data conversion to a suitable format (cbor / json / etc.)
- Multiple storage facilities for reading
- One storage facility for writing

Architecture: Storage Facilities: Data Model

Store								
load	callback	store	Store Instance	path	Registry Path	cb	function(Registry Path, Registry Value)	
save_start	callback	store	Store Instance					
save	callback	store	Store Instance	path	Registry Path	value	Registry Value	
save_end	callback	store	Store Instance					

Store Instance	
itf	Store
data	void * (fs_mount etc.)

Architecture: Storage Facilities: Code Example

vfs_store.h

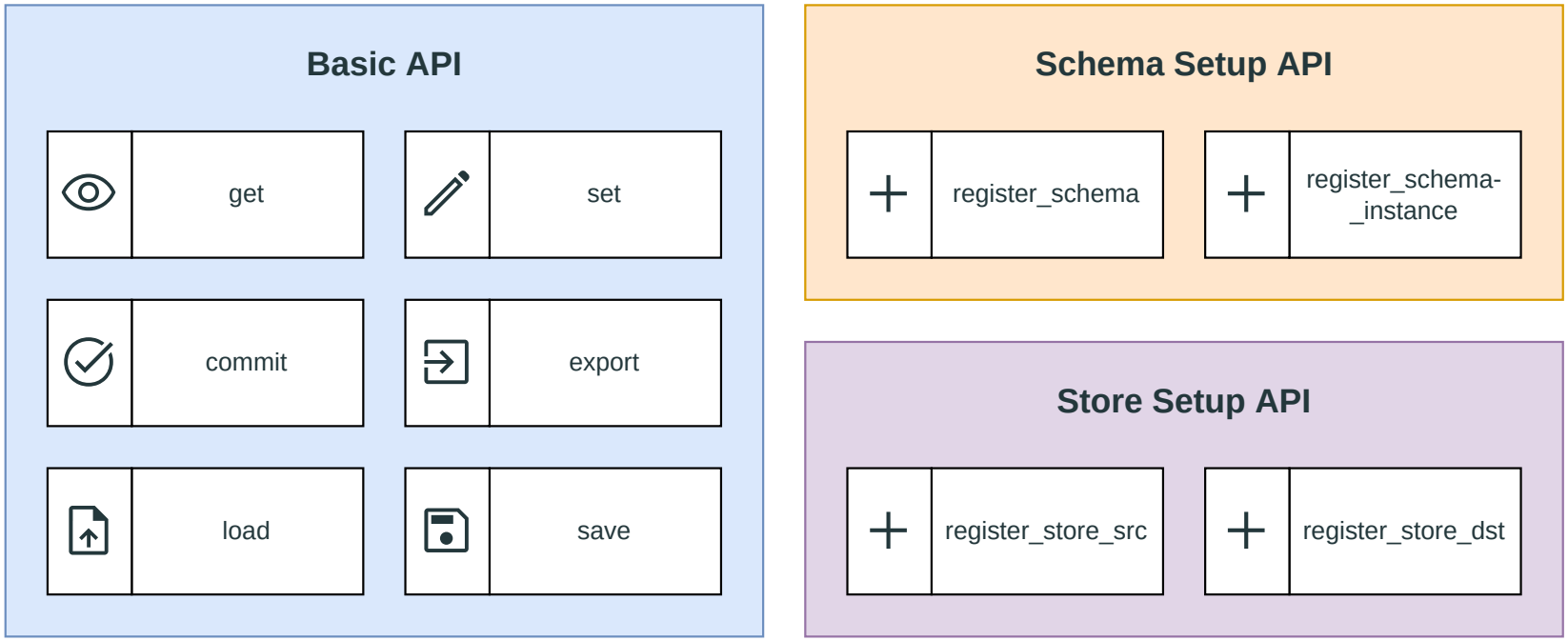
```c
static int load(registry_store_instance_t *store, const registry_path_t path, load_cb_t cb,
                void *cb_arg);
static int save(registry_store_instance_t *store, const registry_path_t path,
                const registry_value_t value);

registry_store_t registry_store_vfs = {
    .load = load,
    .save = save,
};
```

main.c

```c
static littlefs2_desc_t fs_desc = {
    .lock = MUTEX_INIT,
};

static vfs_mount_t _vfs_mount = {
    .fs = &littlefs2_file_system,
    .mount_point = "/sda",
    .private_data = &fs_desc,
};

registry_store_instance_t vfs_instance_1 = {
    .itf = &registry_store_vfs,
    .data = &_vfs_mount,
};

int main(void)
{
    fs_desc.dev = MTD_0;
}
```

API

API

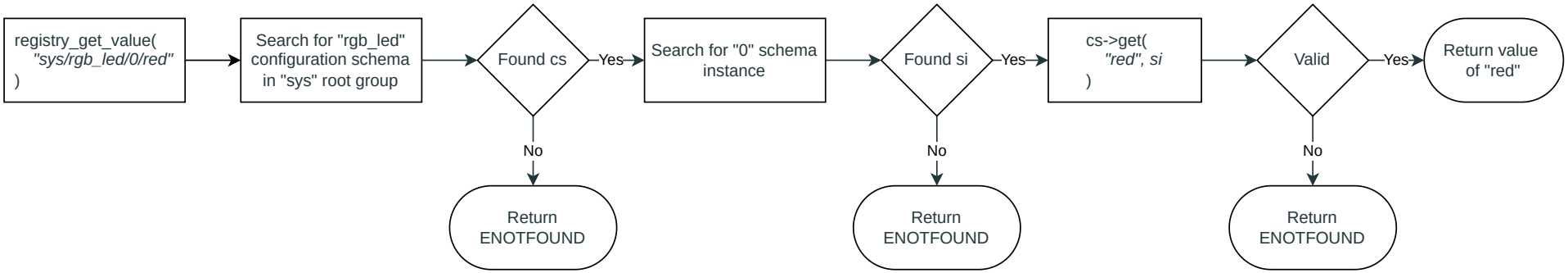

Basic API
get
set
commit
export
load
save
Schema Setup API
register_schema
register_schema-
_instance
Store Setup API
register_store_src
register_store_dst

API: Get

Api

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

Flow

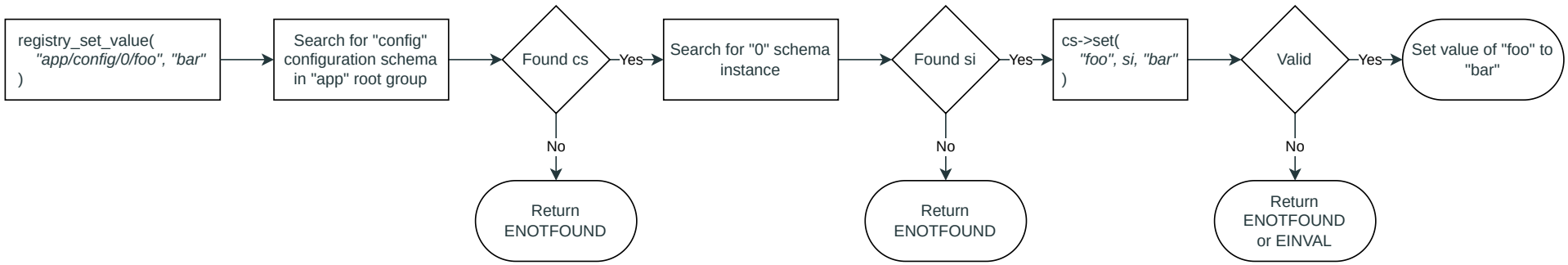


API: Set

Api

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

Flow

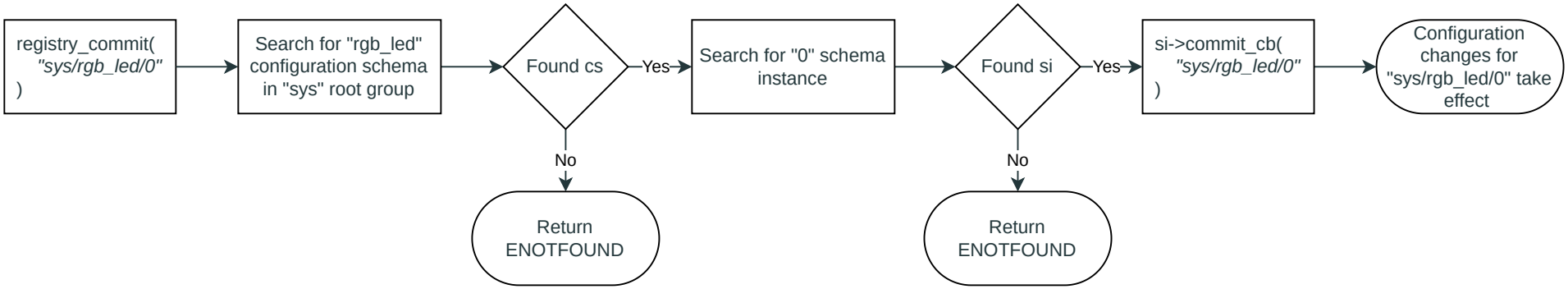


API: Commit

Api

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

Flow

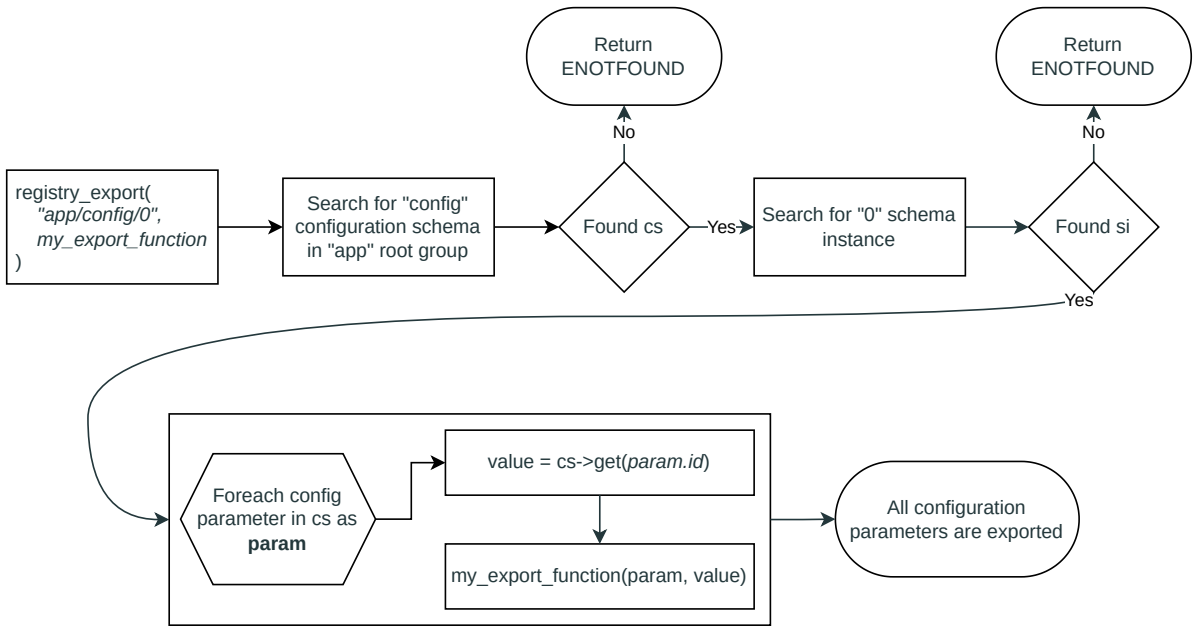


API: Export

Api

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

Flow

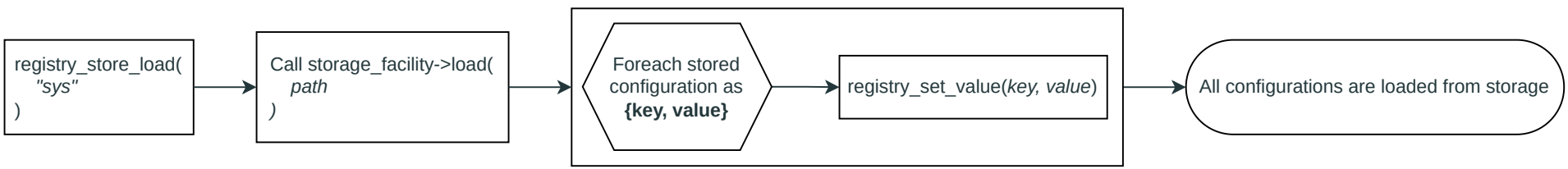


API: Load

Api

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

Flow

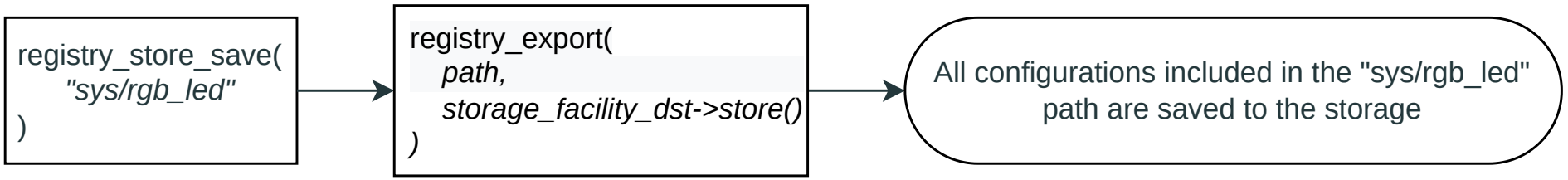


API: Save

Api

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

Flow



API: Setup Schema

Name	Returns	Parameters	Description
register_schema	int	root_group_id: Root Group ID	Register configuration schema
register_schema_instance	int	root_group_id: Root Group ID; schema_id: int	Add instance to configuration schema

API: Setup Store

Name	Returns	Parameters	Description
register_store_src	void	src: Registry Store	Register store to read data from
register_store_dst	void	dst: Registry Store	Register store to write data to

Reasoning behind certain API decisions

Reusable schemas vs. per module schemas

Reusable

- Less redundant work for same purpose modules
- External configuration managers don't need a mapping for each module
- Slightly higher effort at first (Schemas need to be specified carefully to avoid future breaking changes)

Per Module

- Easier implementation for module developers

Integer- vs. String Path

Integer Path

- Smaller payload (except for very short strings)
- Easier to process
- Straightforward to work with through documentation or smart management tools that harness the "name" and "description" fields

String Path

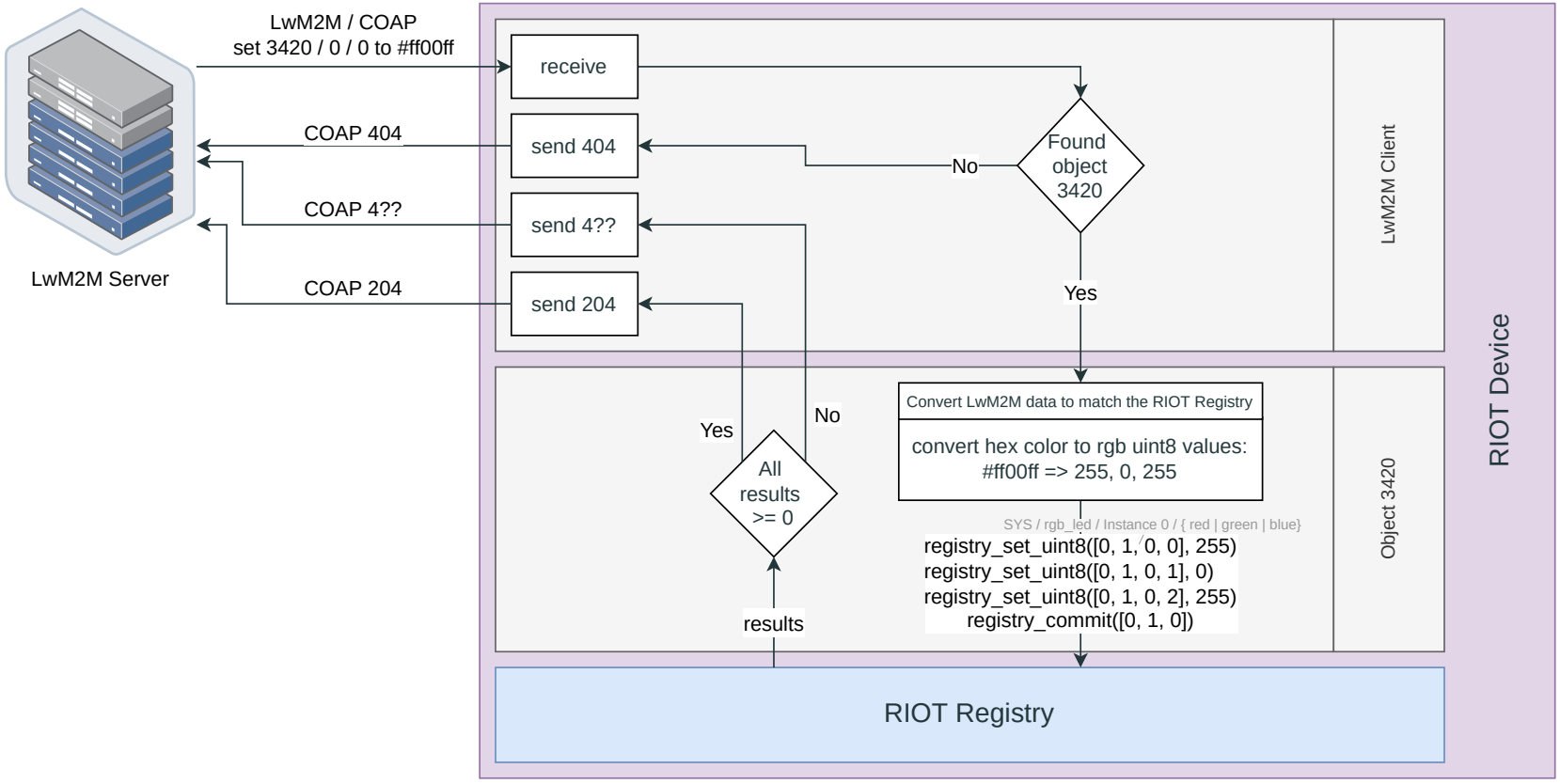
- Possibly human readable (even without documentation)

External Configuration Managers

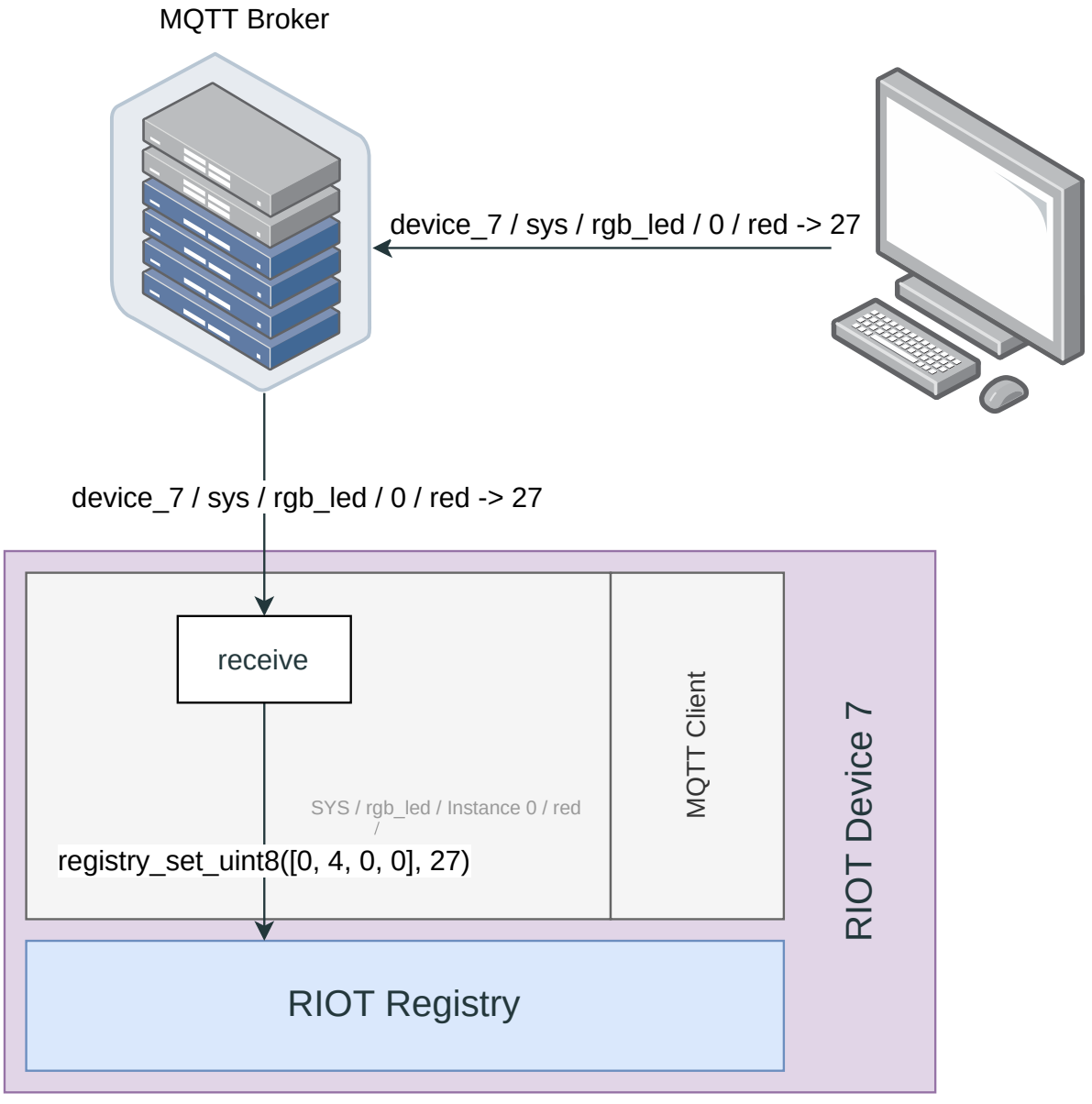
LwM2M Integration

- Modern IoT protocol for remote configuration management
- Specifies object models (schemas) for common configuration scenarios
- Client server model
- LwM2M server knows data structures of specified objects (Object-Model)
- LwM2M client connects itself to the server and communicates which objects it implements
- Configuration Parameter Identification:
  Integer Path: object_id/instance_id/parameter_id

LwM2M Integration



MQTT Integration



Future Work

Future Work

Next

- Specify system schemas (e.g. rgb_led, IEEE802154)
- Expose registry through CoAP API
- Expose registry through MQTT API

Next Next

- Integrate system schemas into modules (e.g. rgb_led for ws28x)
- Implement LwM2M integration for all relevant object models (e.g. 3420 $->$ rgb_led)

Live Demo