Daniil Marinec

Code Maintainability Practices: Effect on the Porting Effort

Metropolia University of Applied Sciences

Bachelor of Engineering

Degree Programme in Information Technology

Bachelor's Thesis

8 May 2025

Abstract

Author:	Daniil Marinec
Title:	Code Maintainability Practices: Effect on the Porting Effort
Number of Pages:	38 pages + 1 appendix
Date:	8 May 2025
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Smart IoT Systems
Supervisors:	Keijo Länsikunnas, Senior Lecturer

This study discusses code maintainability and porting practices and defines the effect they have on each other. Theoretical sources were used to study the topic from an industry point of view, and a code porting project was carried out for a company called Convergens Oy with consideration of existing practices to reinforce the defined relations.

The case project allowed the review of the utilisation of maintainability practices, which benefits and downsides they bring, and how their application affects the working and code porting process. The theoretical study covered such maintainability practices as code modularity, APIs and HALs, automated SW testing, single-responsibility principle, code refactoring and SW/HW integration. All the studied practices were evaluated against their potential or practical application in the case project in light of the SW structure and code analysis.

In conclusion, this study defines the effect that following the studied guidelines has on the code porting process, which focuses on legacy code reuse for a new system design. Furthermore, the study discusses how following or excluding certain maintainability guidelines in practice affected the case project. Their downsides and benefits are also presented in the context of the development of a device aimed for mass-production.

Keywords: code porting, code maintainability, refactoring, API, HAL, legacy code

---

Contents

List of Abbreviations

API:	Application Programming Interface
ARM:	Advanced RISC Machines. A family of RISC instruction set architectures for computer processors.
BLE:	Bluetooth Low Energy
CPU:	Central Processing Unit
FW:	Firmware
HAL:	Hardware Abstraction Layer
HW:	Hardware
I/O:	Input/Output
I2C:	Inter-Integrated Circuit. Synchronous serial communication bus.
LED:	Light-Emitting Diode
MCU:	Microcontroller Unit
PWM:	Pulse-width modulation
RAM:	Random-access memory
RGB:	Additive colour model, which uses red, green and blue as primary colours

RISC: Reduced instruction set computer. A computer architecture designed to simplify the individual instructions given to the computer to accomplish tasks.

SW: Software

TDD: Test-Driven Development

UART: Universal asynchronous receiver-transmitter

1 Introduction

Modern embedded software (SW) development projects often involve legacy code porting, which brings new challenges for SW developers. The case project discussed here, carried out for Convergens Oy, a company based in Helsinki, included porting legacy C code to a newer microcontroller unit (MCU) model based on Advanced RISC Machines (ARM) architecture. This, in turn, raised a question of how the effort required for porting the code could be reduced. This study aims to define a set of guidelines, aiming to reduce the effort required for any present or future code porting actions.

In the embedded systems SW development field, it is important to reduce project costs. Thus, defining an effective set of code maintainability practices would reduce future workload and a significant effort would not be required for the company to stay active.

Section 2 studies existing practices of code maintainability and porting improvements, focusing on their relation to reducing code porting effort in embedded systems. This is followed by a practical code porting example from the case project and challenges faced, described in Section 3.1. The following sections are studying the impact of the application of the researched code maintainability and porting practices during the execution of the case project, identifying which practices were applied and what effect they had on the work effort.

2 Study of Practices

2.1 Code Maintainability Practices

2.1.1 Maintainable Code

Throughout SW development history, maintainability was always one of the biggest concerns of SW developers to the point of creating numerous guidelines

and practices with focus on developing maintainable SW. Maintainable SW is characterised by the effort that is required for its modification, e.g. feature addition, behaviour rework, code refactoring. Additionally, code maintainability tends to “degrade” overtime by itself as hardware (HW) capabilities increase, as well as new programming standards, practices and documentation methods emerge. [1 pp. 12, 30; 2 pp. xiii-xiv.]

Embedded systems tend to further increase the maintainability issues as code in such systems may be deployed for years without a single change to it, which often results in a new team tasked with maintaining the system when the need in change appears. Moreover, embedded systems require SW to access hardware-specific (HW-specific) registers and modules, while some embedded systems impose strict restrictions on code size, which adds another set of challenges for SW design. [1 pp. 18-23; 3.]

Code maintainability affects, and in turn, is affected during the SW development process. Actions that generally increase SW maintainability are listed below:

- Code documentation and commenting
- Dependency breaking
- Separation of responsibilities
- Putting code under tests

In contrast, the following actions decrease SW maintainability:

- Feature addition without design modification
- Introducing untested code to the system
- Reworking or adding code without significant understanding of the system

All the listed actions are interrelated. A higher level of code documentation increases the general understanding of the system during the development process making further reworks or feature additions easier [1 pp. 4, 8-9, 37]. Dependency breaking, aside from increasing code clarity, is also required to put the code under tests, which greatly reduces the risk of introducing untested code [2 pp. 20, 76]. Meanwhile, the design must be flexible enough to allow

effortless addition of new features; otherwise, it is considered insufficient [2 pp. xiii-xiv, 224-225].

To write maintainable code, a programmer must ensure that the written code is tested and documented, and that it has a clear separation of responsibilities between its parts. All three tasks can be dependent on each other in case of automated tests and documentation, which are most useful if included in code design, but can also be kept separate, if done manually. Nevertheless, even manual documentation and testing would benefit from higher separation of responsibilities.

2.1.2 Application Programming Interface

To separate responsibilities in the code, it must consist of separable parts. In object-oriented languages like C++ or Java, parts of the program are represented by classes. While in procedural languages like C, the separability depends on the code modularity. In both cases it is important to keep the responsibilities clear and simple, Listing 1 contains an example of a function with concrete responsibility.

```
/**
 * @brief Erases the whole nv storage.
 */
void nv_erase_storage(void) {
    int res = 0;
    // Erasing nv-storage page-by-page
    for (uint32_t addr = NV_STORAGE_ADDRESS;
         addr < NV_STORAGE_END;
         addr += FLASH_PAGE_SIZE) {
        res += int_flash_page_erase(int_flash_address_to_page(addr));
    }
    syslogNoData(SYS_DEBUG, !res ? "Erased NV-storage"
                                 : "Failed to erase NV-storage.");
}
```

Listing 1. Non-volatile storage erase function.

The function in Listing 1 is responsible for erasing the non-volatile storage allocated for the custom use through the non-volatile storage interface. Instead of accessing device flash directly, the erase function makes use of the “int_flash_page_erase” and “int_flash_address_to_page” functions, which

contain the implementation for MCU flash access and serve as a part of a simple interface for device flash access. In fact, the “nv_erase_storage” function itself is a part of a bigger interface for accessing the special user non-volatile storage flash partition. Such “interface” design allowed the separation of device flash access details from the user storage access interface. Thus, the example function is indifferent to device flash access changes, consequently making it more portable.

The “interface” design presented in the Listing 1 follows the main practice for achieving separation of responsibilities — application programming interface (API) creation. The API is distinct from the application code as it is responsible for certain functionality — accepting defined inputs and producing outputs. Effectively, the API is a movable and reusable encapsulated block of functional code with a distinctive purpose, which can be maintained as a separate module and used to accelerate SW development. [1 pp. 23, 30-31.]

The API serves as an abstraction of the software components, which provide access to them by other SW engineers or by the original programmer. The API is characterised as being a distinct-purpose detachable module, which is dependent on the lower-level code. [7 p. 1.]

Listing 2 shows an example of the RGB LED control interface.

```
#include "rgb.h"

// Skipped definitions and includes
...

void rgb_init(void) {
    // Pins initialization and initial state
    ...
}

void rgb_set_red(uint8_t r) {
    // Parameter is inverted, since LED is active on LOW.
    pwm_set(RGB_MATCH_R, r ^ 0xff);
}

void rgb_set_green(uint8_t g) {
    // Parameter is inverted, since LED is active on LOW.
    pwm_set(RGB_MATCH_G, g ^ 0xff);
}

void rgb_set_blue(uint8_t b) {
    // Parameter is inverted, since LED is active on LOW.
    pwm_set(RGB_MATCH_B, b ^ 0xff);
}

void rgb_set(uint8_t r, uint8_t g, uint8_t b) {
    rgb_set_red(r);
    rgb_set_green(g);
    rgb_set_blue(b);
}
```

Listing 2. Partial RGB LED control interface API.

In Listing 2 the deeper level pulse-width modulation (PWM) API is hidden behind RGB interface functions. “rgb_init” function initialises PWM to a known state and “rgb_set” functions provide an interface for interacting with PWM API in RGB colour value terms, hiding the details of PWM library access. If changes in HW cause the PWM interface to change, the change is only required to PWM calls in RGB interface, without affecting the code that handles the logic of RGB LED control through the RGB API.

The benefit of introducing an API to procedural code, like C, is invaluable, as it creates a distinct module, which, in most cases, can be detached with a minimum number of required modifications, thus, reducing the effort for introducing new changes to the system. Albeit less impactful, introducing an API to the object-oriented code like C++ or Java also increases the code modularity. Nevertheless, in certain cases the API can lead to unnecessarily complicated code and convoluted procedures for its modification. [2 pp. 199-207.]

2.1.3 Single-Responsibility Principle

Another important practice for separating responsibilities is reducing the number of responsibilities of each part of the application to a single responsibility. This practice mainly revolves around making every function purpose described by its function name similar to the example in Listing 3.

```
/**
 * @brief Checks the crc by the given data pointer and length, comparing it
with the given crc.
 * @param ptr [in/out] Pointer to the memory location.
 * @param length [in/out] Length of data to check.
 * @param crc [in] Crc to compare the result to.
 * @return true - match, false - mismatch.
 */
static bool checkCrc(uint8_t * ptr, uint32_t length, uint32_t crc) {
    uint32_t calccrc = startCRC32;
    while (length > 0) {
        int size;
        WD_FEED();
        if (length > FLASH_BUF_SIZE) {
            size = FLASH_BUF_SIZE;
        } else {
            size = length;
        }
        calccrc = crc32(ptr, size, calccrc, CRC32_NO_INVERT);
        ptr += size;
        length -= size;
    }

    // Flip calculated CRC.
    calccrc = crc32(ptr, 0, calccrc, CRC32_INVERT);

    if ( calccrc == crc )
        return true;
    else
        return false;
}
```

Listing 3. Data CRC verification function.

The function in Listing 3 abstracts part of the CRC calculation details. For the purpose of a local function, which is indicated by the “static” modifier, its name is appropriate. The function name claims to verify the passed data CRC and keeps the given promise. The only improvement to its name could be the indication that it specifically calculates a 32-bit CRC: “checkCrc32”.

On the other hand, there are functions similar to the one presented in Listing 4.

```
static remoteReturnCode processRemoteFrame(uint8_t *data, int count, uint8_t
command_type, ackPacket ack) {
    remoteReturnCode result_code = remoteSuccess;
    // Omitted initialisations
    if(data[OFFSET_COMMAND] == COMMAND_A) {
        ... // Omitted update handling code
    }
    else if(data[OFFSET_COMMAND] == COMMAND_B) {
        ... // Omitted update handling code
    }
    else if(data[OFFSET_COMMAND] == COMMAND_C) {
        ... // Omitted command handling code
    }
... // Omitted other commands parsing
    else {
        result_code = remoteCmdUnrecognised;
        sendPortNack(result_code); // Unrecognised command.
        invalidframe = true;
    }
... // Omitted additional code
    return result_code;
}
```

Listing 4. Example structure of a complex function for data and command parsing handling.

The function in Listing 4 claims to “process” the received packet. In reality, this function, alongside with packet unwrapping, parses the data inside it, performs the command recognition, and, in some cases, executes the command, as it is the case with the “update” command handling, which handles the update in-place. It is a clear example of the “monster method”, which has several distinct purposes blended into a single function, breaking the single-responsibility principle [2 pp. 289-294].

The single-responsibility principle concerns APIs, classes and other logically separable parts of the program, (as functions,) by the same extent. If the “and” conjunction must be used to describe an API or class responsibilities, it implies that the API or class in question can be separated, and, in the current state, breaks the single-responsibility principle. [2 pp. 260-263.]

Unfortunately, as any other guideline, it must be followed consciously and applied only when it benefits the code maintainability by increasing clarity or modularity; otherwise, it can result in bringing complexity to otherwise simple code [6 pp. 4-5]. The same holds true for every guideline expressed in the work.

Concerning the single-responsibility principle, it is important to follow the spirit of the guideline. Following the guideline is meant to simplify the code modification process; breaking down functions or classes to a single responsibility deems unnecessary as long as it does not benefit the code clarity or modularity. In such cases the strategy is to group the tightly coupled code together. It can be achieved by moving the simultaneously changed functions under a single class or a file [4 pp. 80-81.] The resulting class breaks the single-responsibility principle, although it benefits ease of access and modification by sacrificing the code modularity.

2.1.4 Test-Driven Development

The longstanding practice for producing tested software to this day still is test-driven development (TDD). The main idea of TDD is to create failing tests for non-existing software parts and write software that passes the aforementioned tests, thus, conforming to the requirements and producing tested software without the need to perform extensive debugging and manual testing after producing the software. [2 pp. 88-90.]

To successfully implement TDD in the workflow, the testing strategy must exhibit following traits:

- All tests must be automated. There is little use in testing a framework, the purpose of which is to decrease the manual testing effort, requiring the aforementioned manual effort for performing tests or verifying test results [4, p. 90].

- Tests must be run frequently. Every test should be run at least once a day to catch bugs as early in development as possible [4, p. 94; 5, pp. 334-335].

- Tests must cover both successful and failing conditions, verifying boundary conditions and sanity checks [4, pp. 99-100].

Following the TDD philosophy creates a unique workflow process. An example for code refactoring with TDD is depicted in Figure 1.

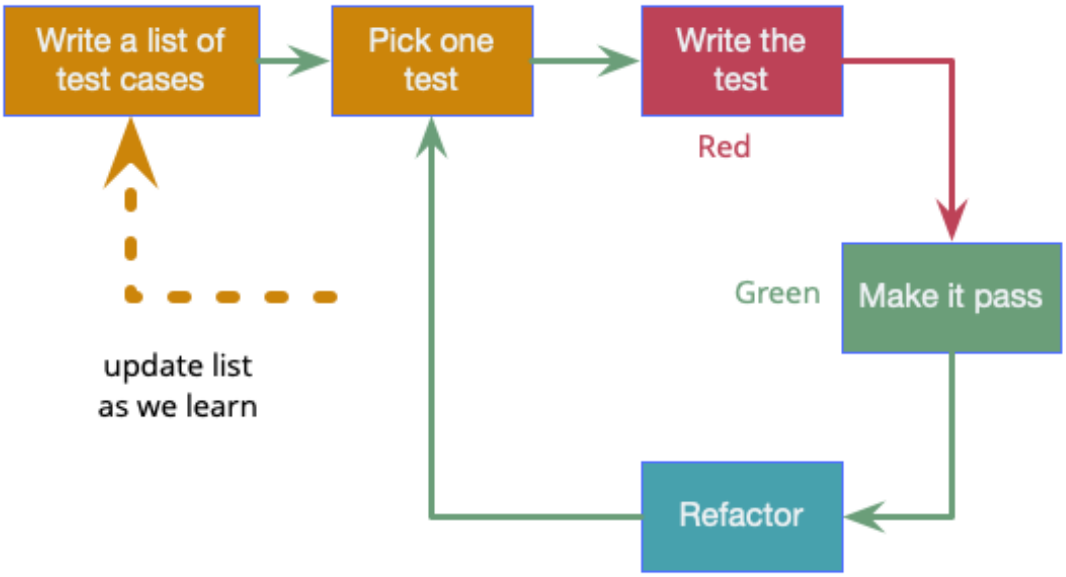


Figure 1: TDD workflow process for code refactoring [8].

The process depicted in Figure 1 consists of one initial and three iterative steps. First, before writing the code, the current implementation and the desired changes to it are assessed, and a list of tests for them is designed. Then the system is put under tests, and tests are written for the new functionality. The next step is to create an implementation that passes written tests and to continue the loop with implementing new tests. Such a workflow produces tested code although there is a slight increase in work time to maintain a test harness and to implement tests. Nonetheless, TDD greatly reduces effort spent on code debugging, which, when done manually, usually takes up most SW development time. [2, pp. 89-90.]

In terms of system maintenance and code porting, the benefit of having an automated testing framework, which can catch introduced bugs during refactoring, feature addition, or porting process, cannot be overstated. The context of embedded systems further reinforces this claim, as it is a rare case to

have an embedded system that is not going to be maintained for an extended period of time.

2.2 Code Porting Practices

2.2.1 Portable Code

Code is considered portable when it requires insignificant effort for transferring the code from one platform to another. Maintainable code does not necessarily imply portable code. Maintainable code can be easier to reuse, but the porting process faces additional challenges compared to those that appear during system maintenance. Such challenges include system HW changes, central processing unit (CPU) architecture, legacy code drawbacks undermining portability and unachievable HW design requirements with maintainable code.

Nowadays, code porting is prevalent in the SW engineering field, ranging from desktop applications to embedded device firmware (FW). Most applications are built on top of existing ones rather than being written from scratch. This was made possible due to the spreading of compilable high-level programming languages such as C, where a compiler took the main effort of translating the language into machine instructions of the specific MCU model. Advancements in the field of embedded systems have greatly reduced the time needed to market the products while the focus of SW engineers has shifted from writing new code to integrating the existing SW for the new HW. [5 pp. xv-xvi, 455-456].

Consequently, alongside with maintainability practices, certain code portability practices emerged in the field of SW development to make the code more flexible in terms of HW integration.

2.2.2 Hardware Abstraction Layer

Hardware Abstraction Layer (HAL) is a layer in the FW code that abstracts HW-level calls, such as CPU register calls. Effectively, it has all the characteristics of

an API although the purpose of an HAL is to create an exchangeable layer between the SW and HW to separate the high-level logic from the HW access. The result of introducing such a layer reduces the effort required for migrating SW from a one HW platform to another, as in majority of cases SW migration requires to modify a HAL for accessing the new HW without the need to modify the higher-level SW. [1 pp. 21-23.]

Listing 5 contains an example of a UART driver “write / put” function, which accesses the MCU register pointer directly for writing into it.

```
void uart_put(int port, char *buffer, int len)
{
      int count;
      int txc = 0;
      int i;
      UART_TYPE *u;
      UART_STATE  *s;

      if((port < 0) || (port >= UART_COUNT)) return;
      if(len <= 0) return;

      s = &state[port];
      u = s->u;
      xSemaphoreTake(s->send_mutex, portMAX_DELAY);
      taskENTER_CRITICAL();

      count = s->tx.count;
      taskEXIT_CRITICAL();

      for (i = 0; i < len; i++) {
            if (count + txc < s->tx.size) {
                  s->tx.buf[s->tx.head++] = buffer[i];
                  if (s->tx.head >= s->tx.size)
                        s->tx.head = 0;
                  txc++;
            }
      }

      taskENTER_CRITICAL();
      s->tx.count += txc;
      u->IER = RXINT | TXINT;
      // after reset we don't get initial tx interrupt so
      // we have to run first write manually
      if(u->SCR == 0) {
            u->SCR = 0xA0 | port;
            uart_transmit(s);
      }
      taskEXIT_CRITICAL();
      xSemaphoreGive(s->send_mutex);
}
```

Listing 5. Example of the UART driver “write” function without HAL.

As indicated by the “u->SCR” register pointer in Listing 5, the UART driver must access the MCU register directly, writing a flag into it, the value and purpose which is indicated in the MCU documentation. In this particular case, the “uart_put” function prepares the MCU for transmitting UART data, and the “uart_transmit” function, (which is also part of the driver,) handles the transmission logic by putting data into the UART transmit register.

In contrast with the example in Listing 5, Listing 6 gives an example of the same “uart_put” function that makes use of the HAL provided by the manufacturer.

```
/**
 * @brief Puts character into UART transmit ring buffer if it has space
available.
 * @param port [in] UART port
 * @param buffer [in] Bytes to transmit over UART.
 * @param len [in] Number of bytes to transmit.
 * @return Count of bytes put into UART transmit buffer.
 */
int uart_put(const int port, char * buffer, const int len) {
    int pos = 0;
    // Take mutex
    xSemaphoreTake(usart[port].mutex_tx, portMAX_DELAY);
    // Give current task handle to port-specific notify_tx.
    usart[port].notify_tx = xTaskGetCurrentTaskHandle();

    while(len > pos) {
        // restrict single write to ring buffer size
        int size = (len - pos) > usart[port].tx_buf_size
                                 ? usart[port].tx_buf_size
                                 : (len - pos);
        // wait until we have space in the ring buffer
        while(usart[port].tx_buf_size -
              RingBuffer_GetCount(usart[port].tx_ring)
              < size) {
            ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
        }
        pos += Chip_UART_SendRB(usart[port].base,
                                usart[port].tx_ring,
                                buffer+pos,
                                size);
    }
    // Give NULL to port-specific notify_tx.
    usart[port].notify_tx = NULL;
    xSemaphoreGive(usart[port].mutex_tx);
    return pos;
}
```

Listing 6. Example of the UART driver “write” function with HAL.

In case of the “uart_put” function in Listing 6, usage of UART HAL’s “RingBuffer_GetCount” and “Chip_UART_SendRB” functions abstracts the use of ring buffer and MCU UART peripheral registers. Thus, in short, the UART

control implementation becomes separable into API and HAL parts, the API depending on the ring buffer and the access abstraction of MCU registers.

The driver in Listing 5 is still considered to be an API, the one that has no HAL layer for the HW registers access. A HAL, effectively, becomes an API for other APIs, abstracting the access to HW, which was the case in Listing 6 where the MCU manufacturer provided functions for HW access in the UART HAL.

As indicated by the example, a HAL can be considered an API. Nevertheless, it has distinct differences from an API due to the limited field of concern. A well-documented and executed HAL significantly reduces the effort developers spend studying the documentation for certain HW and speeds up the development of the target SW. [1 pp. 30-31.]

Nowadays, HALs and APIs for interacting with HW are often provided by the MCU manufacturers, as these provide SW developers with tools for increasing the speed of development, making the choice of the MCU with the provided HALs and APIs more favourable [1 p. 33].

As it is with APIs, HALs have similar drawbacks. When a HAL is poorly documented, untested, or lacking distinct error codes, it can result in increased SW development effort due to the need to debug it. Additionally, a HAL introduces the abstraction that limits the flexibility of the HW access, sometimes resulting in less efficient code, which can be critical for strictly constricted embedded systems. [1 pp. 33-35.]

2.2.3 SW/HW Integration and Co-verification

In modern embedded systems development, a widespread practice is to develop the SW alongside with the HW. Such concurrency allows saving time on SW development as its development is started before the HW design is finished, resulting in less project time or, in other words, less product time-to-market. [5, pp. 401-402.]

Parallel SW and HW development can yield more time savings if the co-verification practice is followed. Figure 2 depicts project time models with and without implemented SW and HW co-verification.

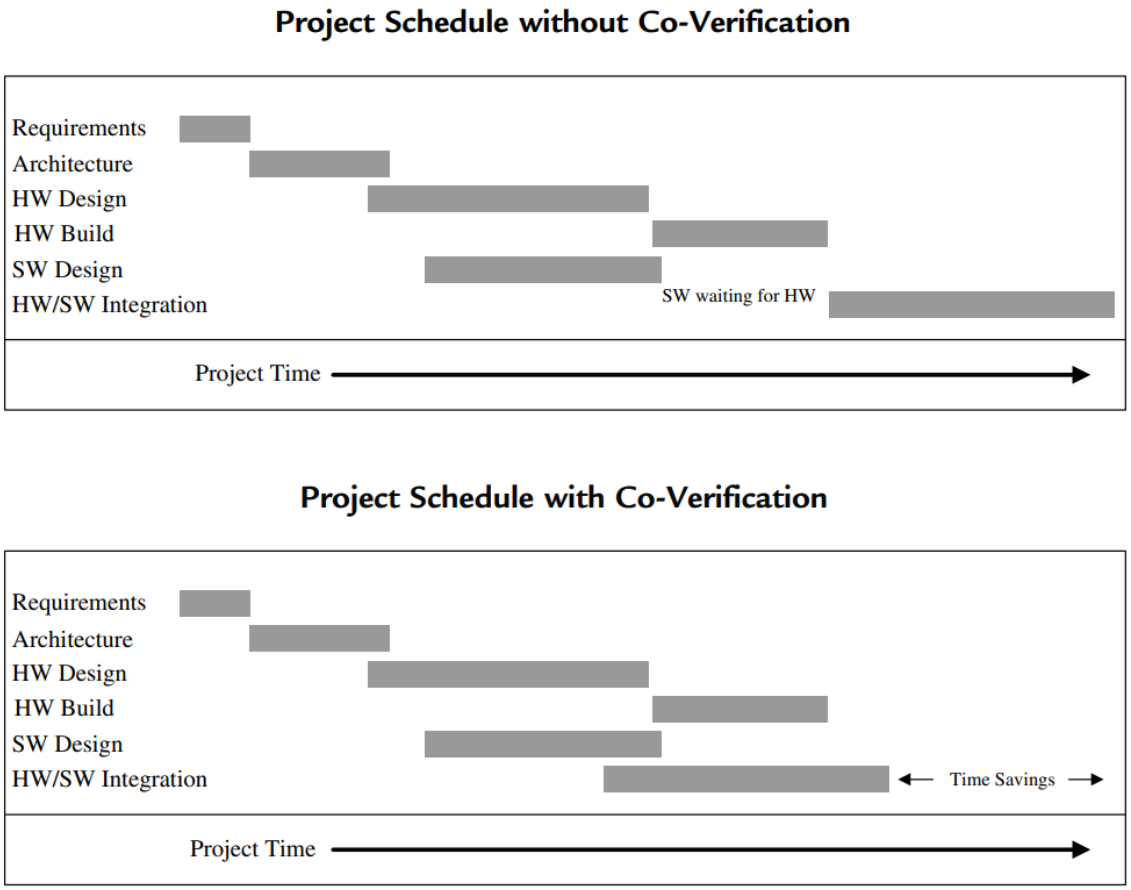


Figure 2: Project schedule without and with co-verification. Adapted from Embedded Software [5, p. 412].

In Figure 2, it is noticeable that co-verification allows to further decrease project time by starting the integration process early, before the HW design is complete. This is achieved by providing SW developers with HW emulators, development kits and HW prototypes to verify the SW against simulated final HW. [5, pp. 410-412] Additionally, a project can have already established and tested APIs, which could be used to test the new or refactored SW against them.

Unfortunately, the time saving benefits of the co-verification process come at the cost of its complexity. Besides HW simulation implications, which are out of the scope of this study, it is important to notice that the aim of verification is to catch design mistakes and imperfections, and their source might be both SW and HW. To effectively tackle SW and HW design errors, SW and HW development teams must extensively communicate with each other to be constantly updated on the state of HW and SW in certain areas. Lack of established communication may lead to searching for an error in SW, while the source of the error could be found in the HW, and vice versa. [5, pp. 413-414.]

Additionally, a HW and SW integration step can bring changes and updates to the SW design, undermining its initial monolithic structure and introducing inconsistencies and trade-off decisions that negatively affect the code maintainability and portability. A common example of such a problem is the code length limitation due to the constricted flash size. Even if the initial design accounted for the constricted flash size, design updates may cause the code length to overshoot the limitation, which calls for a trade-off decision to sacrifice code maintainability and portability in favour of shorter code length.

2.2.4 Preprocessor and Link Seams

In the SW development of embedded systems, it is always the case that parts of the code have to interact with the HW. This interaction results in the requirement of producing HW-specific code, which must be overcome to consider the code portable. Programming languages like C and C++, provide the required tools to facilitate the in-code dependency breaking, allowing to separate the HW-specific code and allow simple switching of HW-specific implementation. Such tools include preprocessing and link seams. [2, p. 33.]

Preprocessing seams in C and C++ make use of the preprocessor directives like “#define”, “#ifdef”, “#if” and similar. Their usage allows for choosing the code to compile during preprocessor compilation stage. [2, p. 34.] An example of using preprocessor directive “#define” is presented in Listing 7.

```
#define FLASH_BASE_ADDR   0      //0x00000
#define FLASH_TOTAL_SIZE  249856 //0x3d000
#define FLASH_PAGE_SIZE   512    //0x00200
#define FLASH_LAST_ADDR   FLASH_TOTAL_SIZE - 1

//Has to be page multiple.
#define FLASH_BUF_SIZE 512

status_t int_flash_page_erase(int page) {
    unsigned int enabled;
    status_t res = -1;

    // Check if we erase a valid page.
    if (((FLASH_PAGE_SIZE * page) > FLASH_TOTAL_SIZE) || page < 0)
        return -1;

    // Disable interrupts.
    enabled = disable_all_interrupts();
    WD_HAND_FEED();

    //Erase the sector.
    res = FLASH_Erase(&_flash_state, int_flash_page_to_address(page),
FLASH_PAGE_SIZE, kFLASH_ApiEraseKey);
    WD_HAND_FEED();

    // Restore interrupts.
    restore_interrupts(enabled);

    // Report result.
    return res;
}
```

Listing 7. Usage of preprocessor directive “#define” in a flash page erasure function.

The function “int_flash_page_erase” from Listing 7 uses “FLASH_TOTAL_SIZE” and “FLASH_PAGE_SIZE” definitions to make its implementation indifferent to possible MCU flash size and pagination. Thus, the example function can be adapted to a different HW by simply changing the definitions values to the corresponding flash HW.

Thus, preprocessor seams are used for grouping HW-specific code, allowing for an effortless switch between different implementations. The downside of this practice is the increased code complexity and the need to, effectively, maintain multiple programs simultaneously.

Another possible solution to the HW-specific code problem is the use of link seams. Generally, their usage, benefits and downsides are similar to C and C++ specific preprocessor seams. A project directory can contain several

implementations of the same function and be configured to link only a certain implementation depending on the HW the build is made for. [2, pp. 36-40.]

2.3 Code Maintainability and Porting Practices' Relation

2.3.1 General Relation

It is often the case that various code maintainability and porting practices relate to each other. Many preprocessor directives can undermine code clarity and increase complexity but improve HAL design, consequently improving API design, which in turn would positively affect the SW and HW integration process. This is one of the many examples of their relation.

First and foremost, reusable code has to be maintainable. The easier it is to understand and modify the code, the easier it is to port it to different HW. Mainly, achieving better code maintainability and portability requires modifying the code in a particular way and with the certain goal in mind.

2.3.2 Code Refactoring

The code porting process implies legacy code reuse. This suggests that, to a certain extent, code behaviour is preserved while integrating it with the new HW. This goal often requires performing the SW refactoring. Refactoring is a process of changing the internal SW structure while preserving its observable behaviour [4 pp. 53-54].

SW refactoring can be done in a single step when it is obviously needed; though, a more effective refactoring strategy is refactoring code as part of the SW development process. When introducing a new feature or fixing a bug, it is a viable option to refactor the existing code, making future feature additions or bug fixes easier to handle in this particular part of the program. It may seem as additional work, but the benefit of the improved SW structure is obvious when

there is a need to revisit refactored code, as the workload for additional changes is decreased. [4 pp. 57-62.]

Although the code reuse is valuable and should be generally applied, when possible, in some cases legacy SW may contain critical bugs, which undermine its refactoring value. In such cases it may be more beneficial to write the required code from scratch rather than attempt to study and refactor the existing implementation. A general practice in such situations is to review the existing code and identify whether it contains key components that still may be reusable after putting reasonable effort into refactoring to detach the components from the code. Reusing key legacy code components and replacing unusable components may prove to be more effective than rewriting everything from scratch or refactoring the legacy SW. [4 p. 66.]

General reasons for refactoring the SW include SW design improvements, optimisation, code length decrease and codebase studying [4 pp. 55-57; 2 p. 3]. SW maintainability benefits from refactoring done for design improvement, while optimisation and code length reduction refactoring goals often appear during bug fixing or the porting process.

Refactoring for SW design improvement includes removing duplicate code, breaking up responsibilities to increase modularity, and regrouping data and code into clumps that share the simultaneous change requirement [4 pp. 77-81]. Named actions increase code clarity, cohesiveness and structural integrity, allowing for quicker code study and modification. During legacy SW porting, programmers can resort to SW design improving refactoring to benefit their understanding of the codebase, speeding up the porting process, meanwhile improving the SW design and benefitting the future SW maintenance [4 pp. 56-57].

On the other hand, refactoring for optimisation or code length reduction is not aimed to benefit the code structure; on the contrary, such refactoring can increase code complexity and result in convoluted code that is optimised for execution in certain circumstances. Nevertheless, it is a necessary step during

the embedded SW porting process, as in most cases it is constricted to be executed in a certain environment, which must be taken into account and appropriately carried over.

2.3.3 Overlapping Practices

Although goals can sometimes conflict, maintainability and porting practices often overlap. Most notably, keeping code modular and separable is beneficial for both codebase maintainability and portability. HW-specific components can be moved to separate detachable modules and replaced when the need arises. Code modularity, in turn, improves code clarity by allowing the codebase to be studied and maintained in independent pieces. More overlapping practices emerge in code commenting guidelines, which are common for maintainable and portable code, with code testing guidelines following a similar pattern.

The first overlapping code porting and maintaining practice is code modularity. It is achieved by breaking the program into specialised parts by their purpose and function: building APIs and HALs, and separating libraries, handlers, or a distinct state machine. The purpose of writing modular code is to enable the interchangeability of distinct code parts without the need to disentangle them from the code. Loose code coupling supports both code maintenance over the period of system lifetime and the porting process by making code reuse much simpler. [1 pp. 9-11.]

An example of a possible modular code structure is presented in Figure 3.

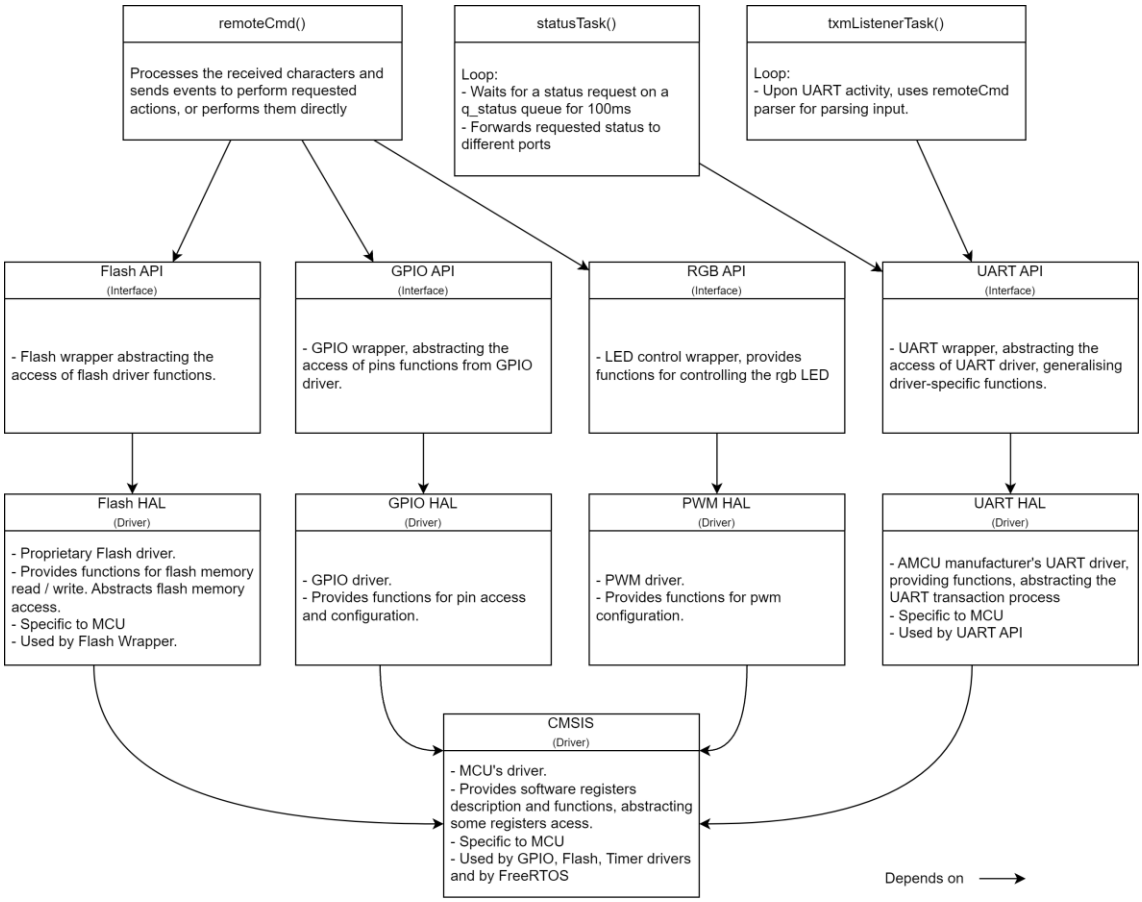


Figure 3: Modular code design example with APIs and HALs.

In Figure 3, the “removeCmd”, “statusTask” and “txmListenerTask” application components use different APIs, which provide interfaces to their corresponding HALs. This is an example of modular code that abstracts the access to HW, making it effortlessly detachable from certain HW implementation, with API interfaces providing the functions that need to be implemented from the HAL side for the SW to operate.

However, modular code design alone is insufficient, as can be seen in Figure 3. Modular code design tends to complicate the code structure, making it harder to navigate for programmers unfamiliar with the codebase. The solution for that is the comprehensive code documentation with descriptive cross references and thinking process explanations. Documented code maintenance can be carried out by non-original programmers without significant loss of work time spent on

code purpose analysis. This also simplifies studying the code structure. [1 p. 121.]

Another crucial practice beneficial for both porting and maintenance alike is SW testing. While TDD supports SW maintenance, it requires a certain test harness. Any working test harness can be utilised during the porting process for performing regression testing, and, preferably, being repurposed for the new system in the future. [1 pp. 257-258.]

Regression testing is a type of testing that is performed on SW to ensure preservation of its behaviour. Such testing is invaluable during the porting process as, when wielded correctly, it can point to undesirable SW behaviour changes, thus preventing bugs during the development process. [1 pp. 268-269.]

Implementing any of the aforementioned practices – code modularity, documentation or SW testing – greatly supports ongoing SW maintenance and enables effective code porting or reuse when needed.

2.3.4 Contradicting Goals and Their Outcomes

A rarer occurrence is when achieving better code portability undermines code maintainability, and vice versa. Constraints of the target platform often cause code porting practices to diverge from maintainability priorities, favouring certain SW design decisions based on deployment requirements rather than code clarity.

One such example is preprocessor and link seams, the extended use of which generally increases code complexity and requires additional effort in SW maintenance to keep the parts of the code which were ambiguated by seams up-to-date. Thus, seams usage requires keeping a careful balance between the urge to write code that covers as much HW as possible and code that is written for the specific platform to reduce SW development effort.

Various code design choice dilemmas often appear during the development process. Sometimes cutting corners is required while developing a certain feature. Sometimes the choice is pushed by the target platform constrictions, and in other cases difficulties are caused by miscommunication or flawed understanding of the legacy code. Problems and their solutions vary from project to project; thus, it is worth examining the process of SW porting on a chosen example.

3 Application in Practice

3.1 Code Porting in Practice

Modern embedded systems projects are often based on already existing systems. In the best case, the new system has a number of insignificant differences or it uses the same or a similar CPU, so that the system SW transfer becomes more of an organisational matter rather than a SW design problem. A more widespread case is the choice of a newer same-family CPU and an update of the system design, requiring a rearrangement of the HW access and structural changes in the SW. In severe cases, code has to be ported across different CPU architectures. There are also borderline cases when the choice between code porting and using an old design as a reference becomes the main topic during project planning.

Taking a look at more widespread porting cases, it is evident that there is a variety of reasons for starting a new system project that changes HW components. Reasons include discontinuation of HW component production, license termination, design improvements, or a change of requirements to HW. Code porting emerges here as a distinct separate action for migrating code from an old system to a new one, not being part of the old system maintenance but clearly dependant on it.

3.1.1 Case Project Description

As an example of code porting action, this study uses a case project which was completed for Convergens Oy. The project goal was to design and build a new thermal printer device alongside with its FW. The printer featured the following:

- Arm Cortex M33 based MCU
- Button and RGB Light-Emitting Diode (LED) user interface
- Serial Universal asynchronous receiver-transmitter (UART) and Bluetooth Low Energy (BLE) communication links
- Thermal printer element MCU controlled by the main MCU via UART
- Main MCU FW written in the C programming language

Figure 4 depicts the shortened main application SW structure of the printer.

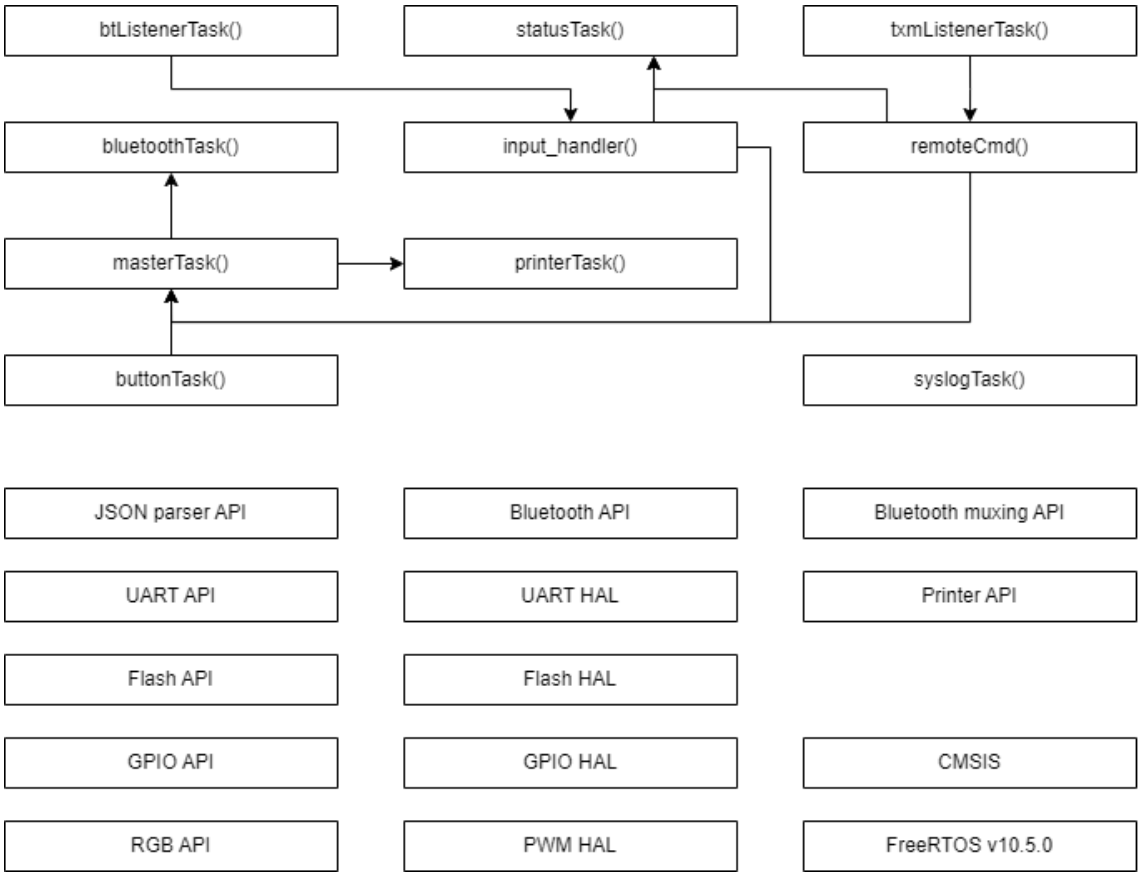


Figure 4: Shortened project SW structure diagram.

The upper part of the diagram in Figure 4 shows the main application components, while the lower part lists the drivers, HALs and APIs. The main application SW components consist of RTOS tasks and extensive input parsing functions, which are executed from one of the listener tasks. Intercommunication of all tasks is performed through events on queues, mutexes and semaphores. A full project SW structure diagram can be found in Appendix 1.

The focus of the SW part of the project was to port legacy thermal printer code to work with a new same-family CPU and design software to accommodate additional requirements and the following HW changes:

- MCU flash reduction from 512 kB to 256 kB
- Touch display changed to button and LED
- Bluetooth module peripheral change

Considering the project goal and HW changes, the following list of required actions was established as the set of SW development tasks:

- Remove touch display controlling functionality from the software and its dependants
- Rework flash partitioning scheme
- Port main software structure
- Update access to MCU registers and peripherals
- Implement new button-LED user interface
- Improve device low power mode (additional requirement)

Every task came with its own set of challenges and had to be done as part of the SW porting process. Some of the tasks affecting the others, such as removal of touch display control, inevitably resulted in software structure modification.

3.1.2 Case Project Challenges

The aforementioned challenge of removing touch display control from the application that affected the software structure was not the only problem faced during the new software design process. Everlasting concern looming over the whole SW design was the flash space reduction, as it was decided that removing touch display logic from the SW would significantly reduce code length allowing designers to use an MCU featuring less flash space.

Another set of challenges was caused by the nature of the legacy SW reuse process. The ported legacy SW selectively complied with maintainability practices, sometimes inconsistently, undermining code portability as a result. A clear example of that would be the legacy SW UART driver, which provided an abstraction layer between MCU register access and the higher-level SW. Unfortunately, the absence of an abstraction layer for accessing digital input/output (I/O) pin registers made it harder to read and port the driver to a newer MCU, which featured a different UART peripheral and manufacturer's HAL for accessing it.

Another example of a challenge arising from legacy SW reuse would be the code that accessed the MCU flash. Legacy SW featured a flash HAL, abstracting the MCU flash access. MCU change resulted in a flash access method difference, calling for the HAL modification for accessing a new MCU flash. Unfortunately, legacy SW modifications underused the flash API, often accessing the flash for reading with direct use of pointers. The new MCU flash restricted access to uninitialised memory, causing the application to crash, while the old MCU returned default unwritten flash value. Thus, all code that read the flash directly had to be wrapped by the flash API calls during the porting process.

3.2 Code Maintainability Practices Application Challenges

3.2.1 Lack of Regression Tests

The ported legacy SW was not put under regression tests for behavioural verification. A constricted project budget made putting SW under tests an unwieldy task, requiring the project SW verification to rely on a manual testing process.

Resorting to manual testing has surmountable drawbacks, which are listed below:

- Increased time consumption. Testing the SW manually requires designing test cases and going through them one-by-one, manually facilitating the needed actions for the test, which considerably increases testing time upon repeating the test.
- Insufficient coverage. Due to the time-consuming nature of manual testing, a SW developer has to resort mostly to functional testing of the recently implemented feature, without performing thorough regression tests of the system SW. Such a course of actions often results in uncaught bugs, which in the best-case scenario are caught during the development process.
- Inconsistency of manual testing. Considering the need to repeat regression tests during the development process, it is important to be able to consistently repeat them. Automated testing is generally more consistent than manual as it runs the script, while the programmer or tester has variance in testing procedure execution. This results in a need to do more repeated tests due to possible mistakes done in testing procedure execution, which further increases the effort spent on testing.

Overall, the lack of automated testing results in limited test coverage and increased effort required for SW development. The effort spent on maintaining a test harness for the system never outweighs the manual testing difficulties in the long term. In case of the printer project, it was decided not to put effort into creating a test harness, as the lack of one for the legacy code required to start it from scratch, which pushed beyond the limitations of the project’s constrained budget.

3.2.2 Embedded Device Environment Constrictions

As was mentioned previously, the constrained nature of the embedded target platform can affect the decision-making regarding SW maintainability and porting practices. The printer project contained an obvious example of that. Listing 8 provides an example of a function that violates the single responsibility principle but had to be designed this way due to constrained random-access memory (RAM) space.

```
uint32_t print_rle_graphics(uint8_t *enc_data, uint32_t size) {
    ... // Data validation and preparation for sending
    // Theoretically, last rle encoded byte should carry enough info to fill
    // the last row. If it is not so, then the last row cannot be sent out.
    while (size) {
        cur_entry = get_rle_entry(&enc_data, &size);
        //wait_ms(50);
        if (cur_entry.count == 0 && size)
            return 3; // Incorrect rle formatting
        else if (cur_entry.raw && cur_entry.count > size)
            return 3; // Raw entries count exceeded the data length.

    // Replicate RLE encoding behaviour to allow processing of flagged data
        if (cur_entry.raw) {
            // The cur_entry.count in here is in bytes.
            uint8_t bit = 0x80;
            rle_entry entry_replica;
            entry_replica.count = 1;
            entry_replica.raw = false;
            while (cur_entry.count) {
                entry_replica.bw = (enc_data[0] & bit) ? 0x80 : 0x00;
                entry_replica.count = 1;
                entry_replica.raw = false;
                if (bit == 0x01) {
                    (cur_entry.count)--;
                    bit = 0x80;
                    size--;
                    enc_data++;
                }
                else {
                    bit = bit >> 1;
                }
                fill_and_send_row(&entry_replica);
            }
        }
        // Filling in normal rle entry.
        else
            fill_and_send_row(&cur_entry);
    }
    return 0;
}
```

Listing 8. The “print_rle_graphics” function for receiving encoded bitmap image data over a serial port.

The “print_rle_graphics” function seen in Listing 8 takes a pointer to raw RLE-encoded data and, while decoding it, and feeds it to the print handling module line-by-line. Thus, the MCU never has to store the full decoded data in the buffer, as such a large buffer could not be allocated in the MCU RAM. This resulted in requirement for simultaneous decoding and data communication processes, which were much easier to facilitate with a single function.

Similar issues were caused by the MCU flash restrictions, requiring simplifying the application structure and refraining from extensive use of the RTOS, which would have allowed separating data acquisition from its parsing. This would have benefited application modularity, but due to the aforementioned restrictions and lack of its separation in legacy SW, it was too costly to implement.

In cases like the ones described in this section, the best practice is to thoroughly document the made decisions and implementation details, for future maintenance to know which decisions lead to this specific implementation and why they were made.

3.2.3 Legacy Code Breaking Single-Responsibility Principle

Budget limitations tend to affect the porting strategy at several levels. The approach to overcoming challenges caused by environment constraints lies the inner layer, which tends to require more in-depth analysis for making the least effort decision. The outer layer involves the decision to refactor the ported code for design improvement to increase SW maintainability. In the case of the printer project, legacy code adhered to maintainability practices selectively, resulting in situations similar to the one presented in Listing 9.

```
/**
 * @brief Processes received framed message
 * @param data [in] array of bytes, containing message data.
 * @param count [in] length of data in bytes.
 */
static remoteReturnCode processRemoteFrame(uint8_t *data, int count, uint8_t
command_type, ackPacket ack) {
    remoteReturnCode result_code = remoteSuccess;
    ... // Frame data structures

    if(data[OFFSET_COMMAND] == UPDATE_CMD && count == UPDATE_CMD_SIZE) {
        ... // SW Update handling code
    }
    else if(data[OFFSET_COMMAND] == UPDATE2_CMD
            && state.packetCount > 0
            && count <= DATA_PACKET_SIZE + UPDATE2_CMD_SIZE) {
        ... // SW Update handling code
    }
    else if(data[OFFSET_COMMAND] == SOME_CMD && count == 1) {
        ... // Process the command
    }
    ... // Skipped commands processing
    else {
        result_code = remoteCmdUnrecognised;
        sendPortNack(result_code); // Unrecognised command.
        invalidframe = true;
    }

    return result_code;
}
```

Listing 9. Shortened function for processing received commands.

In the example presented in Listing 9, the “remoteReturnCode” function was ported from the legacy SW to preserve the command interface. Some effort was put into disentangling command processing from the listener and parser task to have a clear distinction between serial port handling and device state change, which also improved the overall control over the state of the device.

As it can be seen from Listing 9, some functionality, like SW update handling, was left within the function, consequently handling the update procedure from the listener and parser task. Such special cases required special attention and workarounds put into the code to track the device state. In the updating case, the master task had to be notified about the update request and its progress throughout update handling.

Ideally, command processing had to be entirely separated from reception, parsing and interpretation, but the original interface preservation, reuse of legacy code and a constrained project budget required leaving parts of the code

that were hard to disentangle in the “interpreter”. As with all inconsistencies throughout the code design, the key factor for success of this project was the requirement for code comments and documentation, which contained the implementation details and the reasoning behind SW design decisions.

3.2.4 SW/HW Integration Challenges

The printer SW project was carried out in parallel with the HW design and verification. During the SW development the essential part of the verification process was to have a substantial level of understanding of the HW design, as the bug can be caused equally by both SW and HW.

A common example of a HW design problem could be the swapping of transmit and receive UART signals or a wrong resistor connection, which causes failure for SW to communicate over that particular UART port. While in development, the SW implementation of the UART driver and its usage was not reliable. It is also important to remember that HW connections can be wrong. Otherwise, SW developers can keep debugging the UART driver or the code that uses it, while the problem lies in swapped HW signals.

Such integration challenges were tackled by close cooperation between HW and SW development teams. In the case of the printer project, the cooperative workflow was easily established after organising open free-form communication channel between SW and HW teams for discussing observed issues. During issue investigation, it was important to have a method of separately testing HW using verified SW and vice-versa to isolate the problem. From the SW side, for that purpose, it was beneficial to have a ported production test console, which eventually was used in the production testing implementation as well.

Another important point made during the project was the usefulness of link seams for the SW/HW integration process. During the HW revision switch, it was invaluable to introduce linking that allowed switching between platforms by

a single “#define” change, simplifying the process of verifying the new HW design.

3.3 Code Maintainability Improvements Value

3.3.1 Addition of HALs and APIs

One of the factors affecting the code porting effort during the project was the presence of HALs in the legacy code. After starting the code porting process, it became apparent that the old project followed the practice of using HALs only in certain cases. At times, following the practise was undermined by being underused in the project, or it did not affect the porting effort because the code for a particular device feature was useless due to HW change. A clear example of this was the touch display UI code.

HALs, such as the Bluetooth and printer module access interfaces, significantly sped up the understanding of the SW structure. They clearly indicated where the HW access code had to be changed, greatly simplifying the porting of code that relied on these HAL interfaces.

Regrettably, due to the switch from a touch display to the button-and-LED-based UI of the device, the legacy code HAL for the inter-integrated circuit (I2C) and touch display control interface became useless. Notably, the legacy code lacked abstraction of MCU pin access, using direct register pointers. As both the button and the LED must be accessed through a direct I/O interface, it was decided that the API should be introduced for MCU I/O pin access supporting the implementation of the new UI. Both the button and the LED access interfaces were also introduced to the SW.

Additionally, there were two special cases which had to be handled with exceptional care. The first case was the UART driver, which consisted of a HAL and an API mix. This appeared to be an issue, as the manufacturer provided a UART HAL, abstracting the usage of common cases of UART registers for the

new MCU. Due to the changes in the UART registers and how the peripheral is controlled, it was more beneficial to make use of the manufacturer's HAL and turn the legacy UART driver into the UART API, keeping most of the interface intact, but completely changing the implementation. This action required caution and care, determining possible changes in code behaviour.

The second case was the flash driver. As with the UART, it also was a mix of the HAL and the API. Furthermore, as for the UART, the HAL for register access was provided by the manufacturer for the new MCU. Unfortunately, the flash driver had a more significant drawback: it lacked a function for reading the flash, requiring the use of raw pointers to access it. Due to different flash pagination and inability to access the uninitialised flash using a raw pointer, the flash driver required significant rework and expansion to support HAL usage by the manufacturer and to wrap uninitialised memory access.

As a result, the new reworked and introduced APIs follow the practice of using HAL functions, which are often provided by manufacturers nowadays. Alternatively, it is possible to create one's own HAL functions using the MCU specification in place of the existing manufacturer's HAL, which significantly reduces the effort required for code maintenance and porting, as demonstrated by the example of the Bluetooth and printer module APIs.

3.3.2 Introduction of Device SW States

During the project execution, it was planned to create device SW states, which would reflect the current device state and limit or allow certain interaction with the device. An adapted state diagram of the device SW is presented in Figure 5.

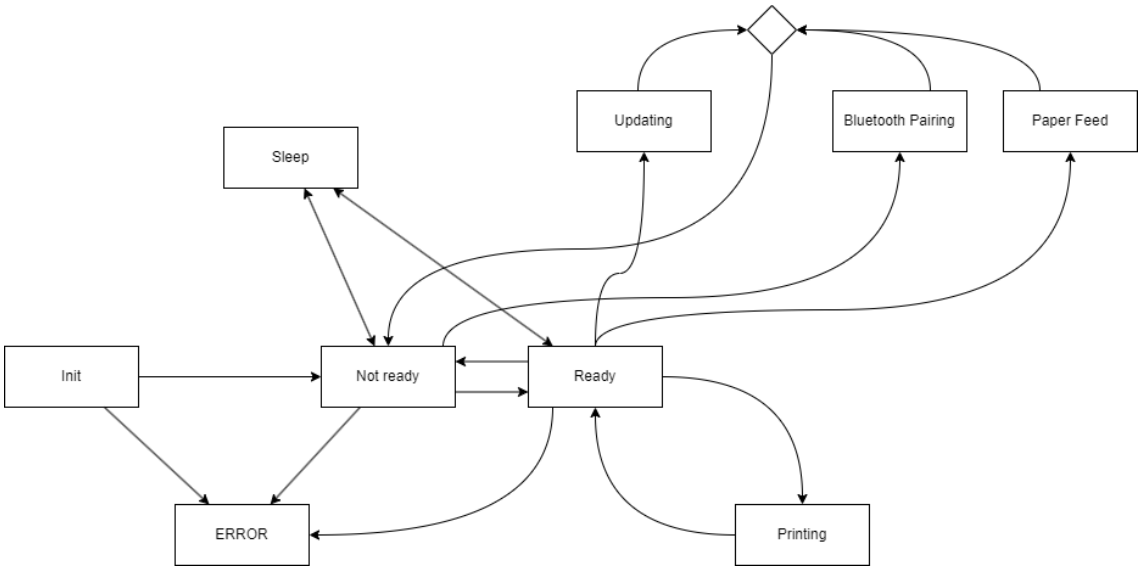


Figure 5: Adapted state diagram of device SW.

The diagram in Figure 5 shows that the device SW has nine distinct states after booting into the application, starting from “Init”. (Bootloader operation and states are omitted for simplicity.) State changes happen automatically, via a serial port request or through specific user button interaction. The state machine of the master task facilitates the device state by controlling the UI, allowing and prohibiting state-dependent commands.

The introduction of such a state machine, separated from input parsing and UI implementation, resulted in a more deterministic device behaviour. Later stages of the project were affected by it in both positive and negative ways. One of the negatives was that the addition of the state machine resulted in greater code length and RAM usage, the shortage of which became apparent during additional feature requests. Flash partitioning had to be revised, and more space was repurposed for application storage.

Nevertheless, the positives of introducing the state machine overweighted the need to review the flash partitioning. The device state machine broke the dependency between the device UI and command parsing, which, in turn, facilitated the rapid addition of newly requested features, concerning both device behaviour and command parsing. Due to the device always being in a

known state, it reduced the effort required for building up confidence in the latest additions, as device behaviour became more predictable with a smaller number of edge code execution cases.

3.3.3 Improving SW Modularity

The improvement of the flash and UART APIs, as well as the introduction of the I/O interface, allowed improving the code structure that handled certain tasks. Port input reading and parsing, MCU flash usage and MCU pin usage all benefitted from the more general MCU HW access functions, thus, facilitating the opportunity for rapid changes to be requested at later SW development stages.

For example, the update packet receiving command, processed in the “processRemoteFrame” function presented in Listing 9, was reworked to separate the flash handling part from the command and response handling. Although the handling remained in the same function, the in-place code separation was beneficial for a further command-interface enhancement request, which required every command to give a response on the port. After separating the flash handling from the port communication code, the addition of the response code was fast and simple. Such a method additionally creates an opportunity for further code enhancement by separating the flash handling part into a separate function, thereby increasing code modularity.

Generally, legacy SW had acceptable code modularity, allowing for code porting conduction in chunks, the only exception being the print handling which was combined with input parsing, UI handling and print formatting. The new implementation tackled this issue, separating out the UI handling. However, input parsing and print formatting proved too costly to fully separate from one another.

3.3.4 Comments and Documentation

As the project was started with insignificant documentation describing the previous implementation, it is a clear example of how the lack of documentation affects the required code porting effort. Starting the SW design with insufficient understanding of the codebase would lead to flawed design. Thus, significant effort was required to create at least partial documentation of the previous SW design. Moreover, careful consideration was needed during the initial porting actions to study the existing codebase operation details before making crucial design change decisions.

Creating and maintaining documentation for the project was a costly and underwhelming effort throughout the initial SW development process, as the lack of thorough knowledge of undocumented and uncommented code tended to cause misinterpretations, undermining the value of the created documentation. Consequently, over time, as more SW was studied and successfully ported to the new HW, this initial documentation was a good reference for both the required project documentation structure and the “pitfalls” in SW understanding that needed to be addressed in the documentation.

Overall, the strategy of refactoring for understanding proved to be an effective approach to this problem. Being in doubt of the original programmer’s intention, it is always worth checking the original code’s operation and carry the code over, improving its clarity and adding comments while keeping the behaviour intact. This ensures the developer’s understanding of the refactored piece of SW, improving the SW design in parallel.

4 Conclusion

This study has looked at how maintainability and porting practices are linked to each other, and now, it is important to give a summary and define a list of practices that affect porting the most. Examination of theory showed that porting practices often overlap with or rely on implemented code maintainability

practices, usually expanding them in the context of embedded systems, where the porting action is most prevalent. Contradicting goals between code maintainability and portability appear only in the context of constrictions introduced by different HW platforms.

The case project examination indicated that the most prevalent issues during project execution were caused by a lack of following certain practises, such as code documentation, which undermines both code maintainability and portability, or missing separation of APIs from HALs, which creates additional challenges for SW porting. Another issue which constantly affected time usage in the project was manual code testing that could have been automated to build up confidence in the SW more easily.

Thus, the list of practices that affect porting the most would include SW documentation and comments, separation of responsibilities (using APIs and HALs,) and SW testing automation. Documentation and comments would have affected the start of the project and improved overall code design without the need to review it during the porting process when the new details about the previous implementation became evident. Extended use of APIs and HALs would have reduced the effort required to port the main application, which was partially true for the case project. SW testing automation, in turn, was required to build up a sufficient level of confidence in the SW with less effort spent on the manual debugging.

Additionally, it is important to note such points as a SW and HW integration process during SW porting action, and the effects of refactoring and other code maintainability practises like single-responsibility principle on the project execution. It is always helpful during SW porting to have the legacy code considering issues and actions and providing tools for tackling integration and refactoring problems with ease, hence underlining the importance of modular code design.

Overall, it is important to follow the spirit of the guidelines and choose wisely in the context of each project whether it is worth to follow them. The case project

example excluded the automated testing guideline, and in the short term was not harmed by this decision. For many porting projects, the prevalent favourable strategy would be to follow as many guidelines for writing maintainable and portable code as possible, as this always benefits the future reworks and refactoring that more often than not are carried out on various embedded systems present on the market.

References

1 Jacob Beningo. 2017. Reusable Firmware Development: A Practical Approach to APIs HALs and Drivers. Springer Science+Business Media, New York, NY.

2 Michael C. Feathers, Robert C. Martin. 2005. Working Effectively with Legacy Code. Pearson Education, Inc., Upper Saddle River, NJ.

3 Momisms [online]. The Ganssle Group. Perfecting the Art of Building Embedded Systems; 2000. URL: https://www.ganssle.com/articles/momisms.htm. Accessed 5 November 2024

4 Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts. 1999. Refactoring, Improving the Design of Existing Code. Addison Wesley Longman, Inc.

5 Jean Labrosse, Jack Ganssle, Tammy Noergaard, Robert Oshana, Colin Walls, Keith Curtis, Jason Andrews, David J. Katz, Rick Gentile, Kamal Hyder, Bob Perrin. 2008. Embedded Software. Elsevier Inc.

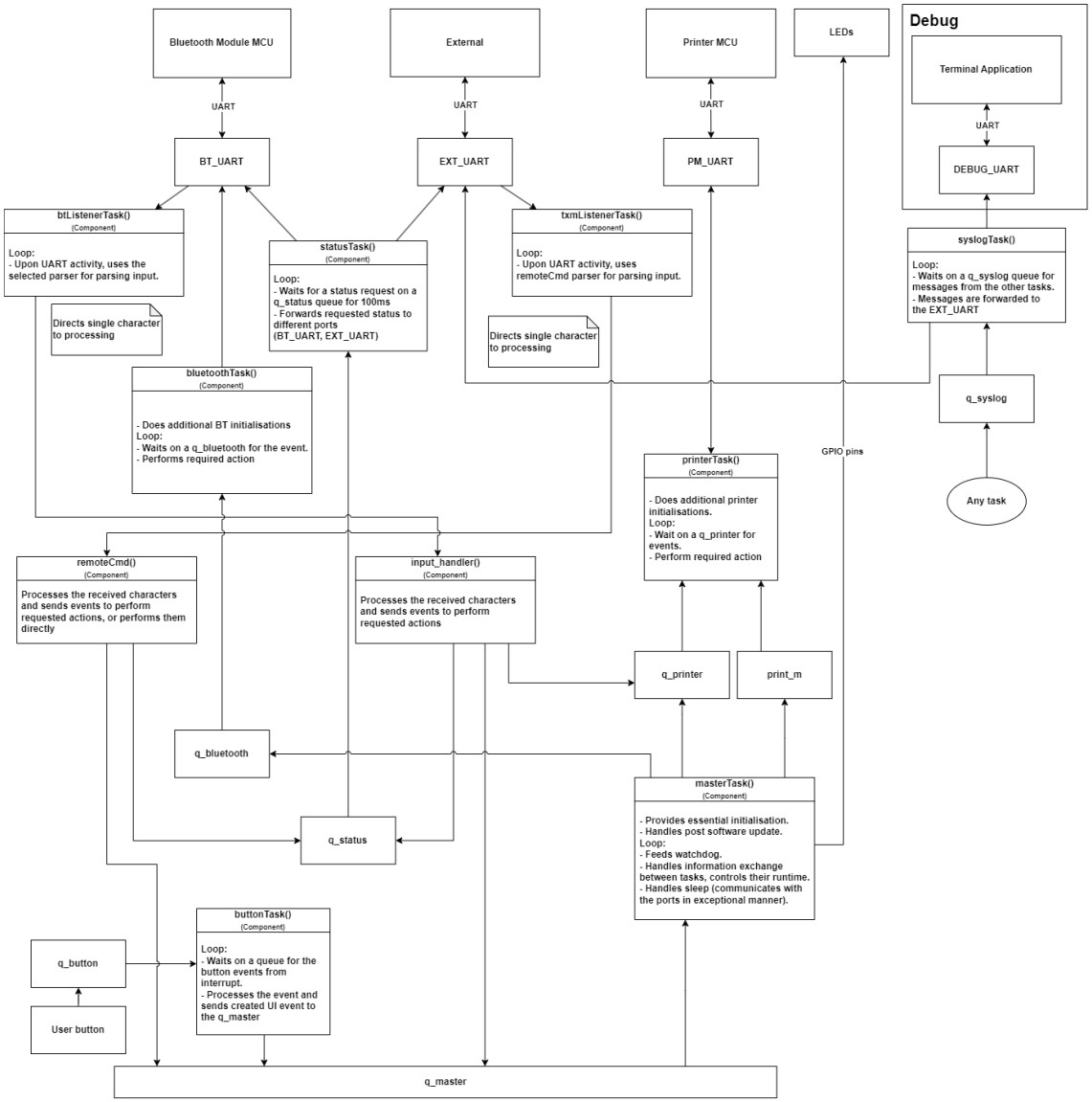
6 Rasmus Storm Bjerreskov. 2023. Developing an Intra-program Messaging Utility in C++. Bachelor's thesis. Metropolia University of Applied Sciences. URL: https://www.theseus.fi/handle/10024/813474. Accessed 7 January 2025.

7 Martin Reddy. 2011. API Design for C++. Elsevier, Inc.

8 Test Driven Development [online]. martinFowler.com; 2023. URL: https://martinfowler.com/bliki/TestDrivenDevelopment.html. Accessed 20 February 2025

Project SW Structure Diagram

Two diagrams presented below separate the case project application SW into logical chunks of code, denoted as components, interfaces and drivers.



JSON parser API
(Component)

- jsmn parser.
- Provides function to parse a json string into the array of tokens.
- Specific to json.
- Used

Bluetooth API
(Interface)

- Bluetooth wrapper abstracts the access of bluetooth module-specific functions, making convinient interface for the application code.

Bluetooth muxing API
(Driver)

- Proprietary bluetooth communication protocol driver.
- Provides function for Bluetoooth message parsing.
- Specific to protocol.
- Used by bluetooth module driver.

UART API
(Interface)

- UART wrapper, abstracting the access of UART driver, generalising driver-specific functions.

UART HAL
(Driver)

- AMCU manufacturer's UART driver, providing functions, abstracting the UART transaction process
- Specific to MCU
- Used by UART API

Printer API
(Interface)

- Printer wrapper abstracts the access of printer-specific functions, making convinient interface for the application code.

Flash API
(Interface)

- Flash wrapper abstracting the access of flash driver functions.

Flash HAL
(Driver)

- Proprietary Flash driver.
- Provides functions for flash memory read / write. Abstracts flash memory access.
- Specific to MCU
- Used by Flash Wrapper.

GPIO API
(Interface)

- GPIO wrapper, abstracting the access of pins functions from GPIO driver.

GPIO HAL
(Driver)

- GPIO driver.
- Provides functions for pin access and configuration.

CMSIS
(Driver)

- MCU's driver.
- Provides software registers description and functions, abstracting some registers acess.
- Specific to MCU
- Used by GPIO, Flash, Timer drivers and by FreeRTOS

RGB API
(Interface)

- LED control wrapper, provides functions for controlling the rgb LED

PWM HAL
(Driver)

- PWM driver.
- Provides functions for pwm configuration.

FreeRTOS v10.5.0
(Component)

- Operating System.
- Provides a method to divide program into tasks, leasing a configurable timeslice for every task to execute.
- Specific to MCU architecture.
- Used by main

The old software uses FreeRTOS v8.0.1
Manufacturer's library contains FreeRTOS v10.5.0
The latest version is FreeRTOS v202212.02