FH Vorarlberg
University of Applied Sciences

Development of a Library for the Detection and Prevention of Errors Concerning Memory Management and Concurrency in C Programs for Microcontrollers Based on ARM Cortex-M Processors

Master Thesis
to obtain the academic degree

Master of Science in Engineering (MSc)

Vorarlberg University of Applied Sciences
Computer Science

Supervised by
Prof. (FH) Dipl.-Ing. Patrick Ritschel

Submitted by
Johannes Koch

Dornbirn, August 2021

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Dornbirn, am 21.08.2021 Johannes Koch

Statuary Declaration

I declare that I have developed and written the enclosed work completely by myself, and have not used sources or means without declaration in the text. Any thoughts from others or literal quotations are clearly marked. This Master Thesis was not used in the same or in a similar version to achieve an academic degree nor has it been published elsewhere.

Dornbirn, on 21.08.2021 Johannes Koch

Kurzreferat

Die Fehlersuche in Softwareanwendungen kann eine große Herausforderung darstellen. Neben dem Wissen bezüglich der Existenz eines Fehlers ist es auch notwendig die Ursache dessen herauszufinden um ihn beheben zu können. Die Suche nach der Fehlerursache kann dabei zeit- und kostenintensiv sein. Im Allgemeinen wird der Teil der Software, welcher vermutlich fehlerhaft ist, analysiert und zur Laufzeit mittels Debugger untersucht. Die Analyse kann dabei durch Instrumentierung begleitet werden um zusätzliche Informationen zu sammeln. Die Analyse wird durch die Existenz unterschiedlicher Fehlerkategorien, welche gegebenenfalls individuell behandelt werden müssen, erschwert. Diese Arbeit beschäftigt sich mit dieser Problemstellung in Bezug auf die Entwicklung von eingebetteten Softwareanwendungen. Der Fokus dieser Arbeit liegt dabei auf der Erkennung und Behebung von Fehlern in Zusammenhang mit Speicherverwaltung und Nebenläufigkeit und soll Entwicklerinnen und Entwickler bei der Fehlersuche unterstützen. Es werden spezifische Eigenschaften der Advanced RISC Machines (ARM) Cortex-M Prozessoren genutzt um Fehler und deren Ursachen zu erkennen beziehungsweise zu verhindern. Ein Beispiel dafür ist die Memory Protection Unit, welche verwendet wird um den Stack-Speicher von einzelnen Tasks in einem real-time operating system (RTOS) zu isolieren. Die gesamte Implementierung versucht so viele Fehlerinformationen wie möglich zur Verfügung zu stellen. Ein weiterer Teil dieser Arbeit ist ein selbst erstellter Speicherallokator welcher Fehler in Bezug auf dynamische Speicherverwaltung erkennt und meldet. Ebenso wurde ein *Eclipse* Plugin umgesetzt, welches den Source-Code durch Asserts instrumentiert um Zugriffe auf Arrays abzusichern. Damit sichergestellt ist, dass die erarbeitete Lösung nicht nur für neue Projekte geeignet ist, sondern auch in bestehende Produkte eingearbeitet werden kann wurde die Lösung in Zusammenhang mit bestehenden, kommerziellen Produkten evaluiert.

Abstract

Debugging errors in software applications can be a major challenge. It is not enough to know that a specific error exists, but the cause of it must be found in order to be able to fix it. Finding the source of an error can be time and cost intensive. The general approach is to analyse and debug the presumably erroneous part of the software. The analysis can be accompanied by instrumentation to gather additional information during the program execution. The analysis is made more difficult by the existence of different errors categories. Each category may need to be handled individually. Especially in embedded software applications, which commonly lack features like process or memory isolation, error detection and prevention can be even more challenging. This is the kind of problem this thesis tackles. This thesis tries to support developers during debugging and troubleshooting. The main focus is on errors related to memory management and concurrency. Specific features and properties of Arm Cortex-M processors are used to try to detect errors as well as their causes. For example, the memory protection unit is used to isolate the stack memories of different tasks running in a RTOS. The thesis tries to provide as much information as possible to the developer when reporting errors of any kind. The solution developed in this thesis also contains a custom memory allocator, which can be used to track down errors related to dynamic memory management. Furthermore, a *Eclipse* plugin has been developed which provides assertions for array accesses to detect and prevent out-of-bound accesses. The resulting solution has been implemented in commercial embedded software applications. This ensures that the developed solution is not only suitable for newly developed applications, but also for the integration into already existing products.

Table of Contents

List of Figures

List of Listings

1 Introduction

During software development, errors can and will occur. Most of them will be found during debugging and testing, but some errors may live long enough to be delivered to an end user. Statistics show that, on average, delivered software contains 1-25 errors per 1000 lines of code (LOC). (McConnell 2004, p. 521 [12])
Because these errors are not detected during development and testing, they can, if they occur, lead to limitations or even to the failure of the software leading to damage (e.g., financial, loss of reputation). The knowledge about the existence of an error is important, but along comes the problem of finding the cause for it. Depending on the kind and source of the error, tracking down and/or the reconstruction of it can prove to be difficult. The research done in this thesis focuses on the problem of detecting and avoiding of specific forms of these kinds of errors mentioned above. Chapter 1.1 highlights the wide adoption of the *C* programming language and Arm processors in the field of embedded software development. The following chapter 1.2 presents the concrete objective of this thesis.

1.1 Motivation

As mentioned in the introduction, software bugs will occur and can have serious consequences. Especially programming languages like *C*, which allow manual memory management and even require it when using dynamic memory, offer additional error potential, in contrast to languages with automatic memory management like, for example, *Java*. Looking at the Common Weakness Enumeration (CWE) Top 25 of 2020 one can see that the existence of errors related to memory management is still relevant. (Mitre 2020, [13]) In the Top 10 of this list are three CWEs which can be directly associated with typical memory management errors in *C*. Ranked on place two is *CWE-787: Out-of-bounds Write*. An error like this can only occur when using programming languages which do not perform bounds checking, like *C* for example. The same applies to place four: *CWE-125: Out-of-bounds Read*. On place eight is a specific case of a dangling pointer: *CWE-416: Use After Free*. This error occurs if manually requested memory has been freed by the developer, but the pointer to the memory location is still being used (See figure 32). Such a problem will only occur when using manual memory management like *C*.

Despite the mentioned properties and potential errors, the *C* programming language is still very actively used. Taking a look at the *TIOBE Index* one can see the ongoing popularity of *C*. Figure 1 shows the popularity of different programming language with *C* and *Java* as the most popular languages.
In embedded software development, *C* is also one of the most used languages. The *IEEE Spectrum*[1] programming languages rating puts *C* on place two after

[1]IEEE Spectrum: Top programming languages (accessed 23 March 2021)
https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2020

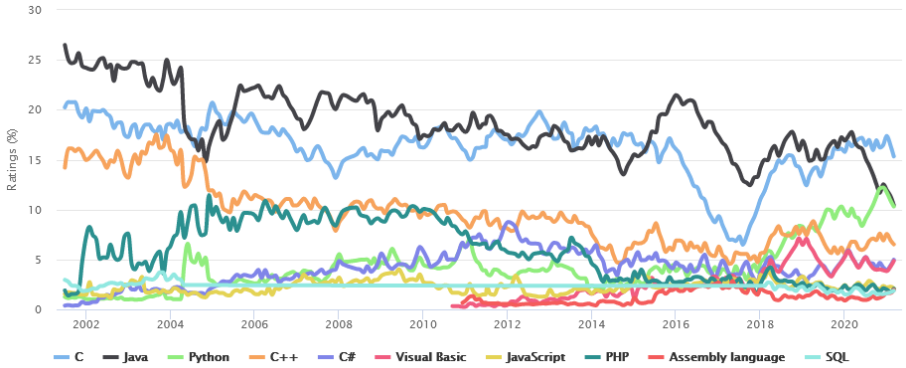


Figure 1: *TIOBE Index* showing the popularity of programming languages over time.

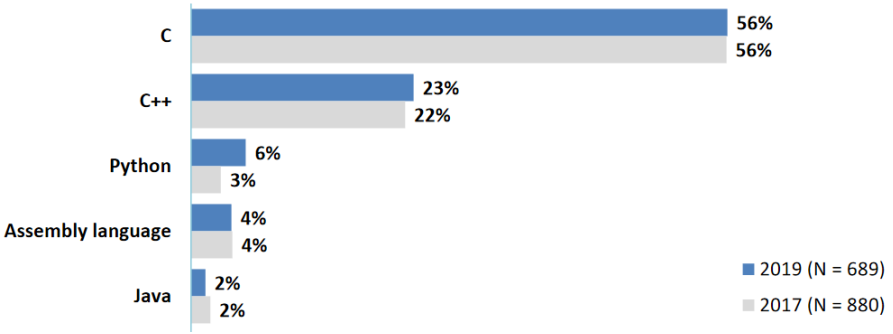


Figure 2: Market share of embedded programming languages according to the 2019 Embedded Markets Study conducted by *AspenCore*

python in the embedded category. The 2019 Embedded Markets Study conducted by *AspenCore*[2] places *C* on first place by a margin of 30% before *C++* and *python*. (See figure 2)

Another dominant part of the embedded market are processors based on architectures from *Arm*. According to the *Arm Strategic Review 2017*[3] *Arm* processor designs were in 34% of all sold silicon chips containing a processor in the year 2016. *Arm* develops and offers three different CPU architecture profiles (A, R and M) to its customers. The A-Profile (Application) is intended to be

[2] AspenCore: 2019 Embedded Markets Study (accessed 30 March 2021) https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf

[3] Arm Strategic Review 2017 (accessed 23 March 2021) https://www.arm.com/-/media/global/company/investors/Arm%20Strategic%20Review%20-%202017.pdf?revision=8473a535-6f7e-4ce5-85fe-0eb6f1f75487&la=en

used for high performance markets such as mobile devices or notebooks. The R-Profile (Real-Time) provides high performance processors for safety-critical environments. The M-Profile (Microcontroller) is intended to be used in embedded systems. [4] The *Cortex-M* series is a processor family that is based on the M-Profile and therefore intended for usage in embedded applications.

Despite the prevalence of the *C* programming language and the *Arm* M-Profile in terms of the *Arm* Cortex-M processor family in the embedded world, there seems to be no (freely available) supporting framework or library to help developers track down memory or concurrency related errors on specifically these platforms. As mentioned before, such errors are still common and can result in expensive and complex debugging efforts.

1.2 Objective

Debugging and tracking down of memory or concurrency related errors often seems non-deterministic, and it can be time and cost intensive to track them down and fix them. The objective of this thesis is to provide support for developers. It focuses on development, debugging and testing support for software written in plain *C* for embedded systems. More precisely, an assistance for the developer to detect or even prevent memory and concurrency related errors during runtime or debugging time will be elaborated. The thesis focuses explicitly on the Cortex-M processor family and will use specific characteristics of these processors to have the best possibilities to detect and prevent errors at disposal. The main focus is on the models: M3, M4 and M7 and therefore on the *ARMv7-M* architecture. Embedded applications are partly built using a RTOS to allow the usage of different tasks that will be run concurrently by the RTOS. The 2019 Embedded Markets Study conducted by *AspenCore*[5] showed that 65% of embedded projects used some kind of operating system. The solution developed in this thesis will also provide support for the usage in combination with a RTOS. Furthermore, the provided features will be used for error detection and prevention.
As described in chapter 2 partial solutions for these kinds of errors already exist in the context of more powerful computer systems. These solutions are mainly possible due to the fact that either some altered form of *C* is used or additional hardware in the form of additional CPU cores or a Memory Management Unit (MMU) is available.
This thesis will provide a solution for the problem mentioned in the context of embedded systems, in the form of the Cortex-M processor family and the usage of plain *C*.

[4] Arm CPU architecture profiles (accessed 27 March 2021)
https://www.arm.com/why-arm/architecture/cpu
[5] See 2

2 State of the Art

There are several ways to detect errors in software. Common to all solutions is the requirement of analysis or tracing to determine where or even why errors occur. Such an analysis can be performed automatically by a specific tool, like a compiler, for example. It can also be performed manually by a software tester or a developer. In general, the different possibilities can be broken down into static and dynamic analysis.

2.1 Static Analysis

Static analysis in general means that the analysis is performed without executing the program concerned. This form of examination can be performed on both the source code of a software and the resulting machine code. The analysing entity processes the source or machine code and tries to confirm the correctness of the software, or at least remove or highlight obvious errors. When referring to static analysis, people tend to assume that tools are performing the analysis automatically. While this can be the case, it is not the only way of performing static analysis. Code reviews, also called code inspections or code walkthroughs, are performed manually by the development team. (Huang 2009, p. 146 [9]) In regard to static analysis tools, a compiler poses a prime example. A compiler will, for example, perform lexical, syntax and semantic analysis. This includes, among other things, type checking. (Aho et al. 2014, p. 5-9 [1]) A failed type check will stop the compilation process and return an error which can, for example, be highlighted for the user by an integrated development environment (IDE). A wide variety of static code analysis tools exist today and many are already part of IDEs, or can be added manually as plugins.

An important aspect in static analysis is the distinction of syntactic and semantic errors. Although compilers can detect syntactic and semantic errors partially, consider the example in listing 1.

```
1  #include <stdlib.h>
2
3  int main(int argc, char* argv[]) {
4    int* pValue = malloc(sizeof(int) * 10);
5    if(pValue == NULL){
6      // Perform out of memory handling
7    }
8    free(pValue);
9    free(pValue);       // Doubel free error resulting in
10                       // undefined behaviour
11 }
```

Listing 1: Double free error: Freeing dynamically allocated data twice

From the point of view of the compiler, the code in listing 1 is perfectly fine. Nevertheless, the code will result in undefined behaviour[6]. Modern IDEs or static analysis tools in general, like a linter, will produce warnings for such a simple double free error. However, such an error may also not be obvious. For example, when two tasks are sharing the same pointer and both try to free the pointer independently of each other. In such a scenario, it may not be possible at all to detect this error using static analysis.

2.2 Dynamic Analysis

There are limitations to static analysis. Static analysis is performed without executing the program to be analysed. Yet there are errors which occur only at runtime and can only be detected during execution of the program. This kind of errors are tackled with dynamic analysis. Dynamic analysis is performed during the execution of a program. This does not necessarily mean that the program is run on a hardware CPU. As we will see in chapter 2.4 it may also be a simulated execution environment. There are various dynamic analysis tools that are dealing with memory-related errors, some of the best known being *AddressSanitizer*[7], *Valgrind*[8], *Dmalloc*[9] and *Intel Inspector*[10]. Most of these tools support different platforms, but are not designed with embedded requirements in mind or they don't provides support for Arm, in this case especially the Cortex-M cores with Armv7-M/Armv7E-M architecture. For example, the Intel Inspector is only available for *Windows* and *Linux* operating systems (10). Another limitation is the resources needed for such tools to operate. This is evident when considering Valgrind's memcheck. Memcheck will slow down the execution of a program by 20-30 times and will at least require 25% of additional memory ([20], [21]).

Considering listing 1 again. We already discussed that the code will compile and result in undefined behaviour. If we take a look at the approach of Dmalloc on detecting memory errors, we see that the default allocation functions (`malloc`, `calloc`, `realloc` and `free`) are replaced with a custom allocator implementation. (See 9) This replacement implementation has to provide memory to the user same as the original implementation. However, to detect a memory error, the allocation functions need to perform additional logic and possible need to manage additional data. (Huang 2009, p. 163 [9]) In the case of Dmalloc, examples for the additional logic and data are: statistics about the current heap, logging and heap integrity verification. Generally, the additional logic is called **instrumentation** and the additional data is called **metadata**. All tools performing dynamic analysis have to instrument the program to analyse

[6] Man page malloc(3) (accessed 09 April 2021): https://man7.org/linux/man-pages/man3/free.3.html

[7] AddressSanitizer (accessed 13 April 2021): https://github.com/google/sanitizers/wiki/AddressSanitizer

[8] Valgrind (accessed 13 April 2021): https://valgrind.org/

[9] Dmalloc (accessed 13 April 2021): https://dmalloc.com/

[10] Intel Inspector (accessed 13 April 2021): https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/inspector.html

in some way. (Nethercote 2004, p. 11f [15]) There are different possibilities to instrument software. Two different examples of instrumentation are explained in the chapters 2.4 and 2.5.

2.3 Instrumentation

In the context of software development, instrumentation refers to the process of adding additional statements to a software with the intent to gather information during the execution. (Huang 2009, p. 163 [9]) This gathered information is used for different purposes like, for example: profiling, error detection, logging and collecting metadata (See Dmalloc example in chapter 2.2). Profiling in this case means the measurement of the memory consumption or the time complexity of the program.

Instrumentation, and program analysis, can be further subdivided into source instrumentation/analysis and binary instrumentation/analysis. The former is performed on source code and when adding new instrumentation, the code needs, at least in the context of compiled programming languages, a recompilation for the instrumentation to take effect. The latter does not need recompilation, as the already compiled binary is instrumented. (Nethercote 2004, p. 12f [15]) It is important to realise that both approaches come with requirements. The source instrumentation is specific to the programming language in use, but it is generally platform independent. Binary instrumentation on the other hand depends on the platform, but not on a specific programming language. (Nethercote 2004, p. 2f [15]; Mußler 2010, p. 12 [14]) In the following chapters 2.4 and 2.5 an example of both kinds of instrumentation is provided. Chapter 2.4 shows an example of binary instrumentation in the form of the dynamic binary instrumentation framework Valgrind. Chapter 2.5 shows an example of source instrumentation in the form of the *GEMS* framework.

2.4 Example: Valgrind

Valgrind is a dynamic binary instrumentation (DBI) framework. A DBI framework provides the foundation to develop dynamic binary analysis (DBA) tools. DBA tools use the framework to instrument the binary. Typical examples of DBA tools are profilers and error detection tools. In the context of Valgrind error detection tools are also called checkers. (Nethercote and Seward 2007, para. 1.1 [16]) While this section will take a closer look at Valgrind, other DBI frameworks exist. The most popular beeing *Intel*'s *Pin* and *DynamoRIO*. All frameworks seem to be under active development as of publication of this thesis, with the latest stable releases from 2020 respectively, 2021.

When developing a new DBA tool, it is much easier to build upon an existing framework like Valgrind instead of creating everything from scratch. A Valgrind tool, for example, will use the available core features and instrument the specific parts of the code which the core provides. (See figure 3)

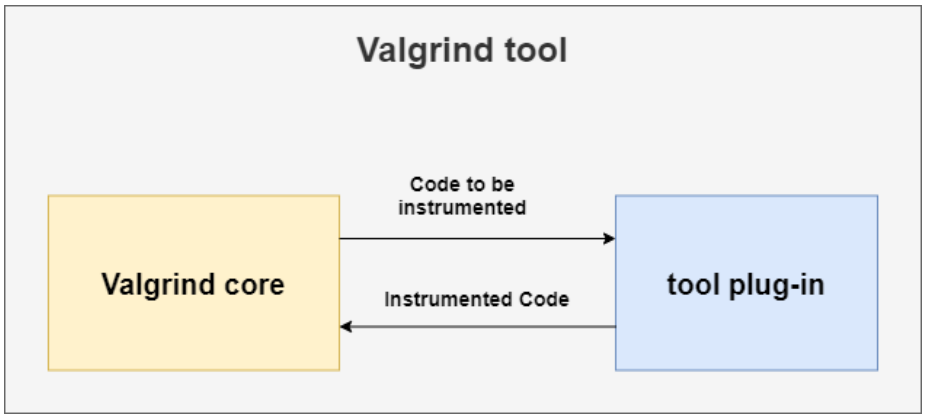


Figure 3: Valgrind core provides the code that will be instrumented by the tool.

2.4.1 Code Instrumentation

The code instrumentation approach Valgrind uses is disassemble-and-resynthesise (D&R). (Nethercote and Seward 2007, para. 3.5 [16]) As the name suggests the provided binary, respectively, the machine code is disassembled and translated into a intermediate representation (IR). The Valgrind IR is RISC-like and therefore each instruction performs only one specific operation. (Nethercote and Seward 2007, para. 3.6 [16]) The mapping to the IR does not need to be one to one. A single machine code statement can result in multiple IR statements. The created IR is then provided to the tool plugin, as shown in figure 3, instrumented as needed and returned to the core. After the translation and instrumentation, the IR is translated into an instruction list and finally into machine code. The generated machine code will be executed and not the original client code. (See section 2.4.3 and figure 4 for a more detailed explanation)
The D&R approach is a unique feature of Valgrind as other DBI's like DynamoRIO or Intel's Pin use a copy-and-annotate (C&A) approach. The C&A approach copies single machine code instructions as they are and annotate each instruction with additional information. The advantage of this technique is the preservation of the original machine code. Tools which depend on low-level information, e.g., if a specific instruction has been used, can profit from a C&A approach. To mitigate this problem in the D&R approach Valgrind's IR supports marker statements that hold information about the original instruction.
A noteworthy advantage of the D&R approach is that the client code and the instrumentation use the same IR, which makes the instrumentation code as powerful as the client code, and optimisation can be performed equally well. (Nethercote and Seward 2007, para. 3.5 [16])

2.4.2 Execution

The starting procedure of Valgrind begins with the initialisation of some subsystems and the loading of the client executable. After the setup of the client stack and data segment, the tool is prompted to initialise itself. Afterwards, additional subsystems are initialised and the translation and execution of the client code starts. The Valgrind core, the tool and the client code reside in the same process and share the address space. (Nethercote and Seward 2007, para. 3.3 [16])
While Valgrind itself runs on the host CPU, the instrumented client program runs on a virtual CPU. Similarly, the Client CPU registers (also called guest registers) are stored, among other data, in the client `ThreadState` which is managed by the Valgrind core per individual client thread. (Nethercote and Seward 2007, para. 3.4 [16]; Valgrind Developers [22])

2.4.3 Code Translation

Due to the D&R approach of Valgrind, the translation from machine code to the IR and back to machine code is complex. The process consists of eight phases, which are presented in Figure 4. The advantage of using a DBI framework is clearly visible. The developer of a tool based on Valgrind only needs to interact with Phase 3 from figure 4 to instrument a binary. The IR that is used for the code translation and instrumentation is expressive, but there are limitations. It can, for example, not express memory allocations and deallocations or other memory state and guest register adjustments done by clients. To mitigate this limitation, Valgrind provides an event system and function replacement/wrapping. (Nethercote and Seward 2007, para. 3.12 [16])

2.4.4 Event System & Function Replacement/Wrapping

The event system is used to communicate changes to guest registers and memory state done by clients. The event system provides different callbacks, which a tool can register for. For example, there are `pre_reg_read` and `post_reg_write` events. This events inform registered callbacks that a register is going to be read by a system call respectively that a system call has written a new value to a register. (Nethercote and Seward 2007, para. 3.12 [16]) This callbacks are essential for tools that use shadow values. Shadow values will not be explained in this thesis. For additional information about shadow values and there usage in *Memcheck* see [16] and [15].

The aspect of heap memory allocation and deallocation is not tracked by the event system. To be able to track these events in a tool, the tool needs to provide a replacement or a wrapper implementation. Valgrind supports function replacement, and therefore a tool can provide an alternative implementation for a specific function. The alternative implementation can call the replaced function and thus offer function wrapping. (Nethercote and Seward 2007, para. 3.13 [16])

2.5 Example: Generic Model-Based Source Code Instrumentation (GEMS)

The Generic Model-Based Source Code Instrumentation (GEMS) framework tries, as the name suggest, to provide a universal instrumentation framework for different programming languages. The developers of the framework defined a core IR which contains the common constructs of all the source languages, as well as language-specific models that contain the distinct differences of each language. Furthermore, a so-called Unified Programming Language Model (UPLM), which is a union of all IR models, is defined. (Chittimalli and Shah 2012, p. 909 [7]) Figure 5 shows the instrumentation process using GEMS. The programs (source codes) are translated into a IR. This IR contains an Abstract syntax tree (AST) which is used to perform the instrumentation. After the instrumentation, the IR is translated to source code, which now includes the instrumentation. The developers highlighted the separation of concerns. The instrumentation does not depend on the programming language. However, this means that translators from source code to IR and from IR back to the source code are needed for each language. When executing the programs after the instrumentation, a language-specific execution library is needed.

The general approach of the GEMS instrumentation is quite similar to the array bound asserter, described in chapter 4.3. The bound asserter is implemented as a *Eclipse* plugin. The Eclipse environment is used to create and obtain an AST of the associated source code. This AST is traversed and, similar to the Probe Insertor in figure 5, the instrumentation code is inserted and reflected in the source code again. The concrete instrumentation in this case are custom assert statements for array accesses. Another similarity to the GEMS instrumentation is the fact that the instrumentation needs an execution library. In case of the array bound asserter, a specific include directive is inserted into the source code files. The user needs to make sure that the needed library is available.

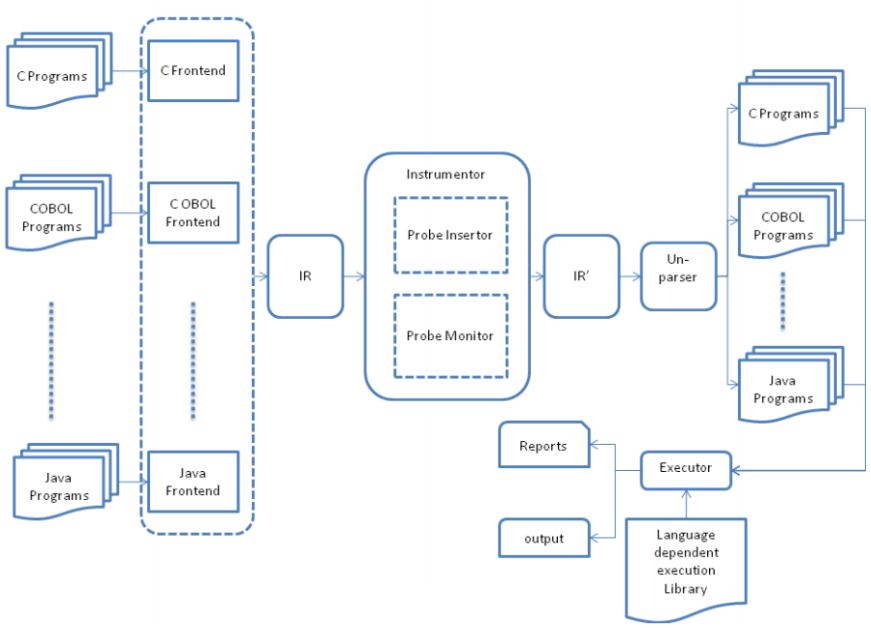


Figure 5: The code instrumentation flow of the GEMS framework. The source code programs are translated into a universal IR, which is then instrumented. After the instrumentation, the IR is translated back into source code. To execute the resulting programs, a language-specific execution library is needed. (Source: Chittimalli and Shah 2012, p. 911 [7])

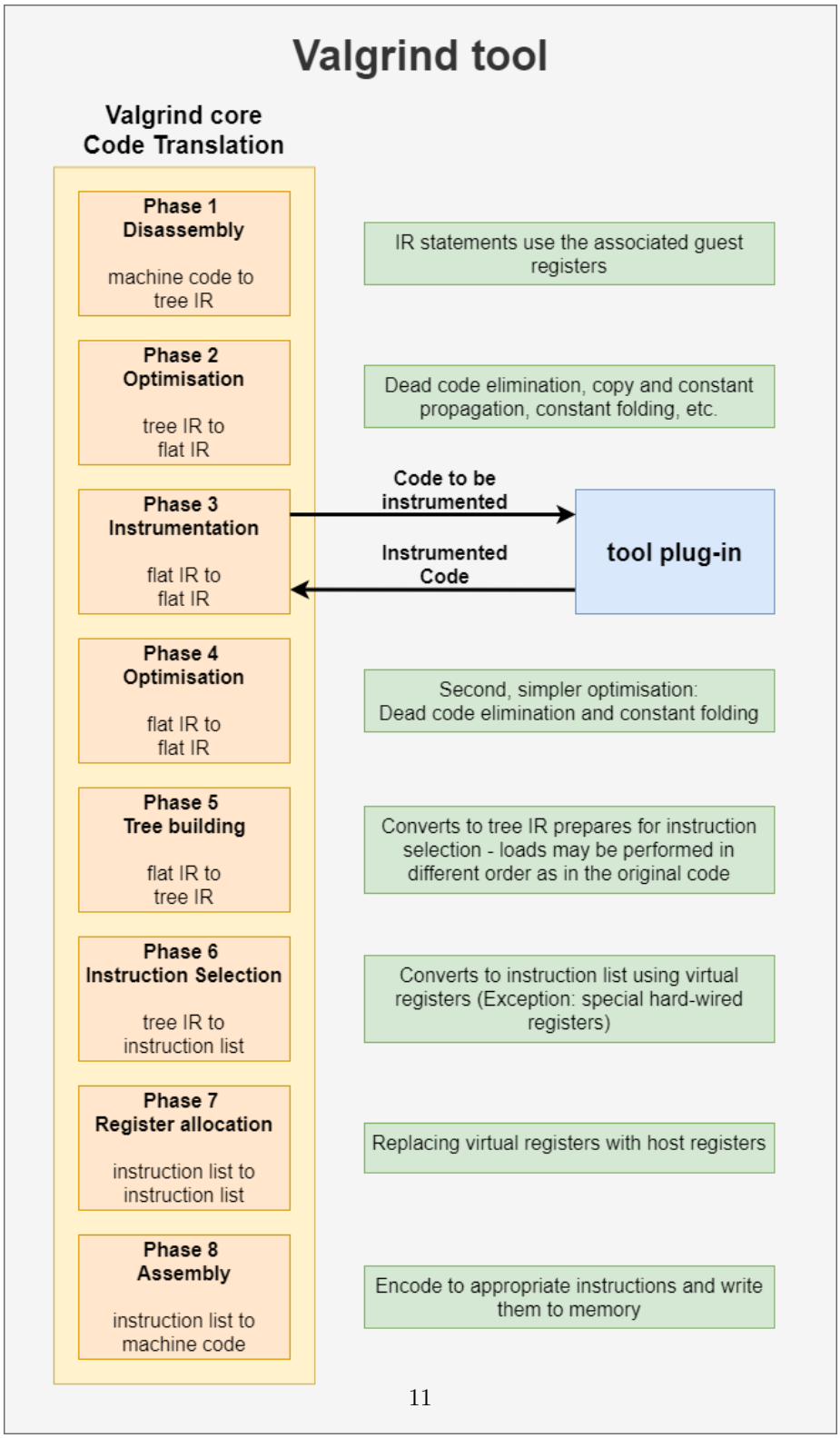


Figure 4: Valgrind translation process. Dead code elimination, constant folding and copy Propagation are explained in the glossary.
(Nethercote and Seward 2007, para. 3.7 [16])

3 Errors in Embedded Software Development

This chapter provides a categorisation of errors in embedded software development. Chapter 3.1 describes the additional challenges of embedded systems in contrast to non-embedded application software. It also gives a general overview of the error categories in embedded software. The following chapter 3.2 describes on which error categories this thesis focuses.

3.1 Classification of Errors in Embedded Software Development

Software development for embedded systems poses additional challenges for developers compared to developing non-embedded software applications based on operating systems like *Linux* distributions, *Windows* or *macOS*. Embedded software needs to implement and control all the hardware drivers which are needed to properly operate the device by itself. Embedded software also needs to process hardware and software interrupts. Hardware interrupts can occur, for example, when receiving a message via the serial interface. Depending on the priority of an occurring interrupt and on the activation of interrupts in general, the associated interrupt handler will be called. In general, interrupt service routines (ISR) should execute as briefly as possible and never block the execution, as this could, for example, result in errors related to timing or task management. If multiple task are needed, the developer needs to configure them properly for the specific RTOS in use. Furthermore, as there is most likely no virtual memory abstraction, the RTOS respectively the developer needs to make sure that the tasks are isolated properly. With these additional challenges and responsibilities also comes additional potential for errors, as well as additional sources of errors. Kim and Huh[10] have developed, based on several additional research papers on embedded software defects, the following table (6). It records their final collection of accumulated and derived embedded software defects. Some defects, like *E11 - Dynamic memory*, apply to programs written in *C* in general. However, error codes like *E7 - Device driver* or *E8 - Hardware interrupt* affect embedded software exclusively.

3.2 Errors Discussed in This Thesis

The different defects listed in 6 are ranked according to different metrics. One of the metrics is called *D+R*. The *D* value represents the degree to which one defect affects other defects listed in the table. The *R* value represents the degree to which a specific defect is affected by other defects listed in the table. The combination of these values as *D+R* value represents the sum of the *D* and *R* value and therefore shows how strongly this defect interacts with other defects. (Kim and Huh, p. 12 [10]) The defect *E3 - Task management* has the highest *D+R* value of all defects. This means that it is a central component regarding defects and a good candidate for debugging defects.

Code	Embedded Software Defects	Definition	Sub-Defects
E1	Wrong logic	Control logic and calculation	Control flow, if, case, loop statements, divided by zero
E2	Wrong function	Function itself defects	Non-reentrant function, incorrect objects
E3	Task management	Concurrent processing error	Deadlock, race condition, task management
E4	Exception handling	Device driver exception handle error	Software exception handling excluding device driver error
E5	Internal software interface	Communication error between software	Internal interface, inconsistent module interface, wrong parameter
E6	External interface	Communication error with the external system	Networking, send and receive packet error, human interface
E7	Device driver	Hardware control device driver	I/O device, I/O port process, I/O device status
E8	Hardware interrupt	The processing routine for hardware interrupt	Non interrupt routine, incorrect interrupt routine, process error
E9	Timing error	Defects that cannot complete in time	Time out, time delay, feedback control error, set time and read time
E10	Data, shared memory	Data and static memory	Data definition, data access, shared memory
E11	Dynamic memory	Defect using dynamic memory	Memory initialization, memory management, resource leaks, memory overflow
E12	Flash memory and file system	Data storage device	Flash memory, storage data save

Figure 6: Kim and Huh[10] have collected and derived this table from several research papers regarding defects of embedded software applications. They extracted these 12 embedded software defects using content analysis procedures. See the source for more details. (Source: Kim and Huh, p. 10 [10])

This thesis focuses on detection or even prevention of errors related to dynamic memory management (*E11*), shared memory (*E12*), out-of-bound array access (*E1*) as well as concurrency related errors (*E3*). (See table 6) One of the biggest tasks when fixing errors is to find the source of the error. Arafa et al. [2] conducted a study in 2017 where participants (intermediate-level developers) tried to fix bugs in a RTOS. The RTOS is similar to one they have already worked with. One of the observations of the study is that over 93% of the participants considered finding the bug to be more difficult, or at least as difficult, as fixing the bug. The evaluation of the study showed that only 63% of the bugs have been located by the participants. (Arafa 2017, p. 2 ff. [2]) To support developers with error detection or even error prevention, the following chapter 4 introduces the *Sheaperd* library. The library is intended to support developers during the testing and debugging process. As finding errors can prove even more difficult than fixing them, the detection and reporting of errors and, if possible, their origin is a key feature of the Sheaperd library.

4 Sheaperd - A Library to Support Debugging and Testing of ARM Cortex-M Devices

The Sheaperd *C* library was developed as part of this thesis. The library is designed to help developers to detect and fix errors and focuses on errors related to memory management and concurrency. It consists of three main modules. *Sheap* is a custom memory allocator which is described in detail in chapter 4.1. The *Stackguard* module and its base module, the Memory Protection module, is presented in chapter 4.2. Additionally, the *Eclipse* plugin provides array bound assertions as well as debug support in the form of an integrated executor view and is explained in detail in chapter 4.3. This chapter describes the general concepts of the Sheaperd library.

When initializing the Sheaperd module, a user can provide a callback function (`sheaperd_assertion_cb`). This function will be called when an assert inside the library fails. The callback provides an error category as well as a text describing the assertion. Listing 2 shows the error categories and the callback registration. The library can be configured using several defines, which are used for conditional compilation. The default settings are defined in the file `opt.h`. This file includes the file `sheaperdopts.h` which represents the user defined settings. Any setting in `sheaperdopts.h` has priority over the default settings.

```
typedef enum {
  SHEAPERD_GENERAL_ASSERT,
  SHEAPERD_ARRAY_BOUND_CHECK,
  SHEAP_INIT_INVALID_SIZE,
  SHEAP_NOT_INITIALIZED,
  SHEAP_OUT_OF_MEMORY,
  SHEAP_SIZE_ZERO_ALLOC,
  SHEAP_ERROR_INVALID_BLOCK,
  SHEAP_ERROR_DOUBLE_FREE,
  SHEAP_ERROR_NULL_FREE,
  SHEAP_ERROR_OUT_OF_BOUND_WRITE,
  SHEAP_ERROR_FREE_PTR_NOT_IN_HEAP,
  SHEAP_ERROR_FREE_INVALID_BOUNDARY,
  SHEAP_ERROR_FREE_INVALID_HEADER,
  SHEAP_ERROR_FREE_BLOCK_ALTERED_CRC_INVALID,
  SHEAP_ERROR_COALESCING_NEXT_BLOCK_ALTERED_INVALID_CRC,
  SHEAP_ERROR_COALESCING_PREV_BLOCK_ALTERED_INVALID_CRC,
  SHEAP_ERROR_MUTEX_CREATION_FAILED,
  SHEAP_ERROR_MUTEX_DELETION_FAILED,
  SHEAP_ERROR_MUTEX_IS_NULL,
  SHEAP_ERROR_MUTEX_ACQUIRE_FAILED,
  SHEAP_ERROR_MUTEX_RELEASE_FAILED,
  SHEAP_CONFIG_ERROR_INVALID_ALLOCATION_STRATEGY
} sheaperd_assertion_t;

typedef void (*sheaperd_assertion_cb) (sheaperd_assertion_t assert,
     char msg[]);
void sheaperd_init(sheaperd_assertion_cb assertionCallback);
```

Listing 2: An excerpt of the Sheaperd library showing the different assertion categories, the callback function prototype as well as the intitialization function.

4.1 Secure Heap (Sheap) - A Custom Memory Allocator

The *Sheap* part of the *Sheaperd* library provides a custom heap allocator implementation. In contrast to common heap allocator implementations, the Sheap allocator contains additional safeguards. These safeguards are intended to detect common errors related to dynamic memory management. Inevitable, these additional safety measures are accompanied by disadvantages. The main disadvantages are the increase of the space and time complexity. These increases are introduced by the additional logic for the error detection and the layout of the memory blocks. The mentioned disadvantages are bearable, as the Sheaperd library is intended to be used for testing and debugging purposes. Besides the additional safety measurements, the Sheap allocator can collect information and statistics, which may help to find the origin of an error.

This chapter will give an overview of the Sheap allocator implementation. The memory allocation and memory block layout is presented and explained in chapter 4.1.1. Chapter 4.1.2 shows how the implementation tries to detect errors at runtime.

4.1.1 Memory Allocation and Block Layout

The Sheap memory block layout is depicted in figure 7. The overhead introduced by Sheap to maintain the heap structure and provide error detection is 16 bytes. The memory block header (and boundary) consists of the aligned size requested by the user, the calculated alignment offset and a cyclic redundancy check (crc) checksum. The size alignment depends on a user-defined value, but will at least be four bytes. If the value of the alignment offset is greater than zero, it can be used to check for possible buffer overruns when freeing the memory block. (See section 4.1.2) The crc checksum is calculated using the data of the memory block header except the checksum itself. (See figure 7) Initially, the Sheap allocator is initialized with a `starting address` and a `heap size`. The implementation will create an initial memory block with a payload size of `heap size - overhead(16 bytes)`. Figure 8 shows the memory after the heap initialization with a heap size of 1000 bytes and a heap starting address of `0x20002800`. The figure shows that an initial memory block with a size of 984 bytes payload (1000 bytes - 16 bytes overhead) is created. The `allocation flag (A)` is zero as the memory is not allocated yet, the alignment offset is zero as the requested size is already aligned to 4 bytes and the crc has been calculated and written to memory. The boundary tag mirrors the header for convenience regarding the coalescing of freed memory blocks and error detection.

Memory Allocation

After the initialisation, the Sheap allocator is ready to use. The general process of allocating memory with the Sheap allocator is depicted in figure 9. The figure mentions the search for the next block of adequate size. There are different approaches to find a block of memory of the requested size. The Sheap allocator does not use any explicit free list to record available memory blocks. The heap is traversed using the start address and the memory block size, which is stored in the header. The Sheap allocator values simplicity and therefore uses a **first fit** approach for finding free memory blocks. The first fit strategy traverses the heap in search of a big enough memory block and uses, as the name suggests, the first one it finds. This approach is generally fast as it searches as little as possible and stops as soon as the first suitable block is found. (Tanenbaum and Bos 2015, p. 192 f. [18]) Nevertheless, it must be mentioned that the first fit strategy also comes with a disadvantage. It tends to split the large blocks at the beginning of the heap first. This can result in lots of small memory blocks in the beginning of the heap which need to be traversed each time a free block is needed, thus resulting in longer searches. (Wilson et al. 1995, p. 30 f. [23]) Another possible approach for searching free memory blocks would be **best fit**. This strategy searches the heap for the best suited memory block in terms of size, with the idea of wasting as little memory as possible. The main disadvantage of this approach is that it is exhaustive in general. It may only stop before traversing the whole heap if a perfect fit is found. (Wilson et al. 1995, p. 30 [23]) These are just two examples of a variety of different approaches.

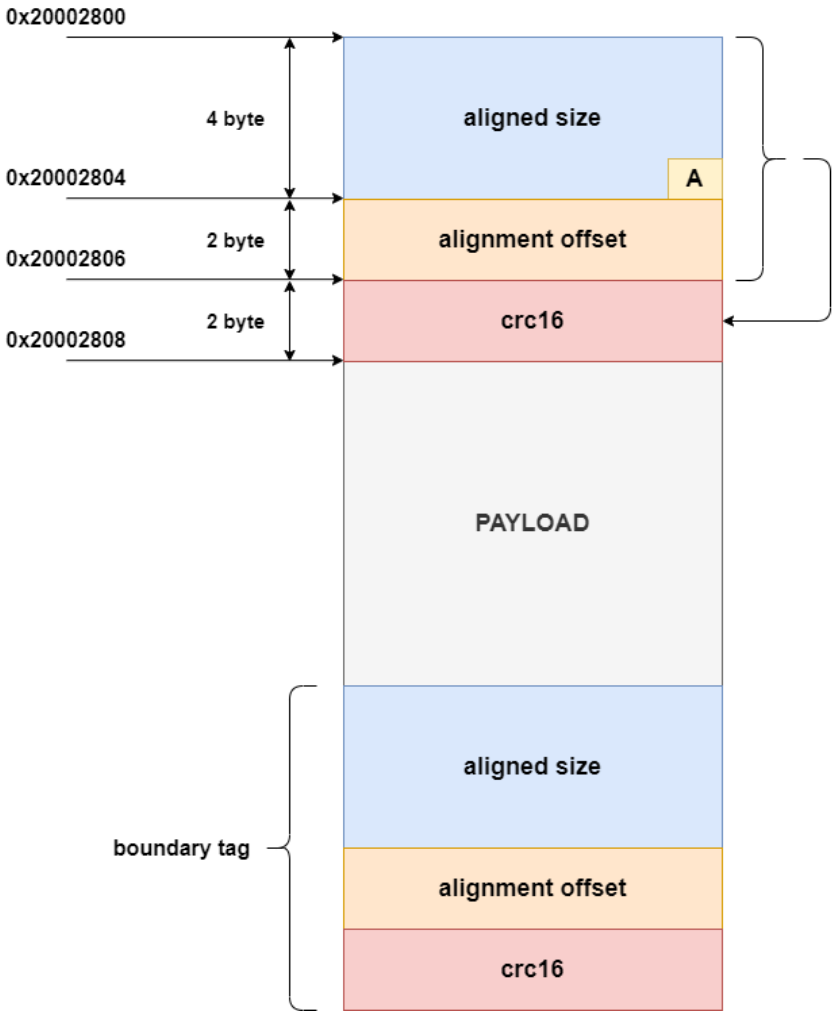


Figure 7: An example memory block of the Sheap allocator located at address 0×20002800. The header and boundary data is used to mark if a block is in use, to navigate through the heap, ease the coalescing of memory blocks and for error detection.

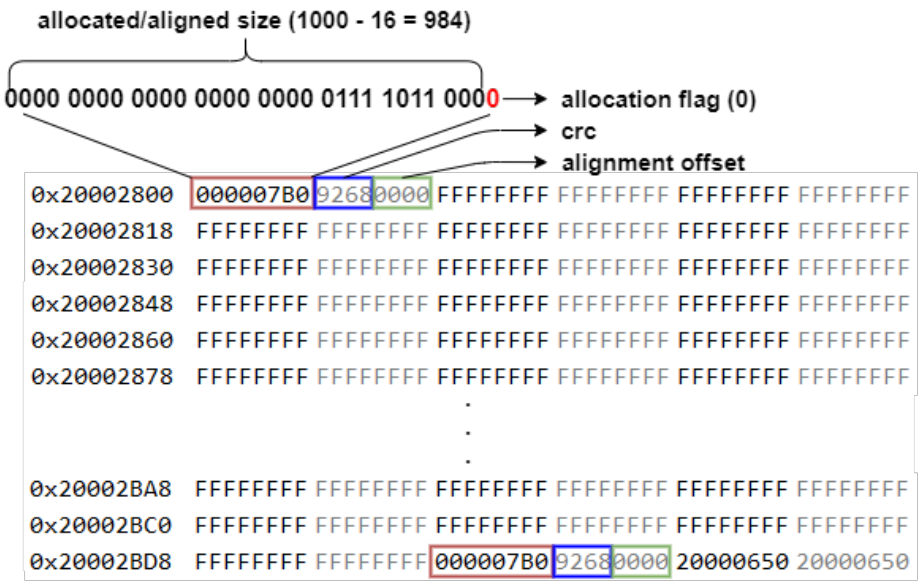


Figure 8: The heap after the initialisation of the Sheap allocator with a size of 1000 bytes. Note that the displayed memory is configured for interpretation as 32-bit integer values. Therefore, the display of the 16-bit crc and the alignment offset values are displayed in the opposite order as explained in figure 7.

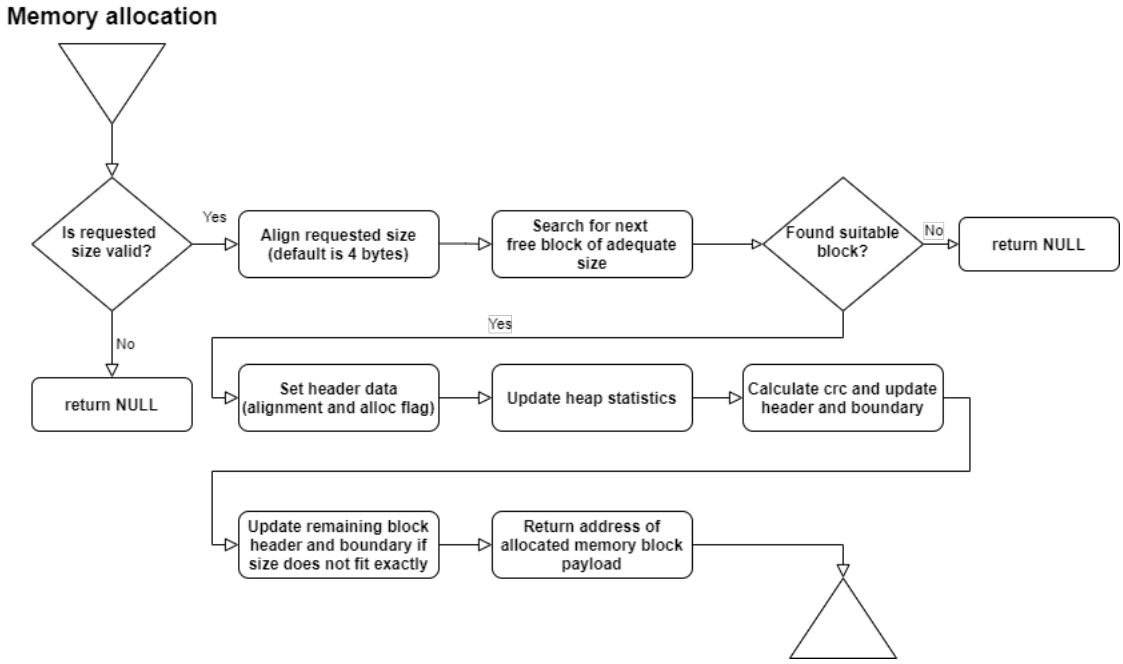


Figure 9: Control flow of the memory allocation in the Sheap allocator. Individual statements like concurrency handling with mutual exclusion have been omitted in this flowchart for the sake of clarity.

This thesis will not go into more detail about other approaches. (See Wilson et al. [23] for more information on the subject) The Sheap allocator can be extended to use additional memory allocation strategies besides first fit.

Memory Deallocation

Memory that has been dynamically allocated also needs to be deallocated or freed again. If memory is not freed, the executing program will most likely run out of memory at some point in time. The general process of freeing memory with the Sheap allocator is depicted in figure 10.

Freeing a memory block itself is as simple as changing the allocation flag (A) from one to zero. However, if that is the only operation that takes place when freeing a memory block, the memory will suffer from fragmentation at some point in time. After the initialisation, one big block of memory exists. (See figure 8) After some memory blocks have been allocated, this one big block has been divided into multiple smaller blocks. Figure 11 shows the heap as initialised in Figure 8 after three allocations of 17, 50 and 150 bytes.

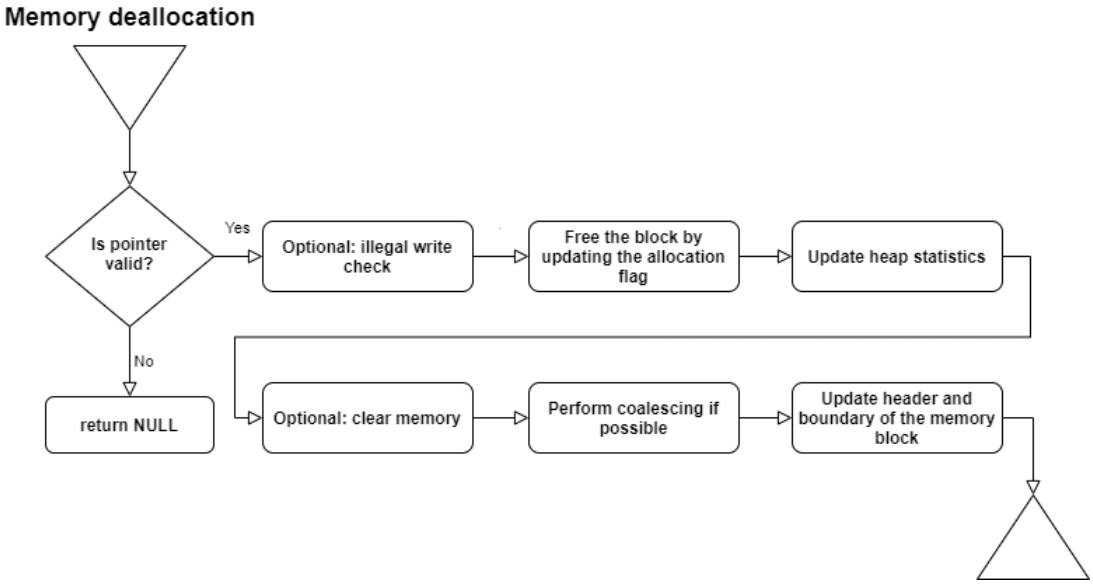
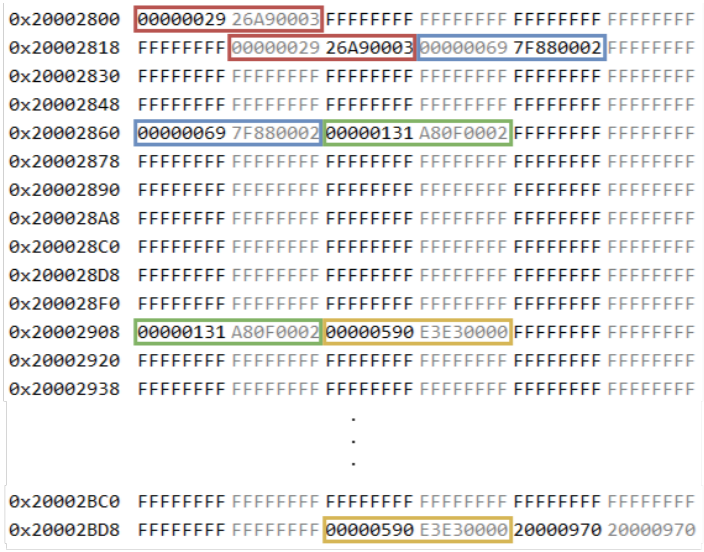


Figure 10: Control flow of the memory deallocation in the Sheap allocator. Individual statements like concurrency handling with mutual exclusion have been omitted in this flowchart for the sake of clarity. Coalescing means merging free adjacent memory blocks to one bigger block. More detailed information about the pointer validity check and the illegal write check is available in chapter 4.1.2.



Memory block header	allocation flag	alignment size	alignment offset	crc
0x00 0x00 0x00 0x29 | 0x26 0xA9 0x00 0x03	1 - true	0x29 >> 1 = 20_{10}	3_{10}	0x26A9
0x00 0x00 0x00 0x69 | 0x7F 0x88 0x00 0x02	1 - true	0x69 >> 1 = 52_{10}	2_{10}	0x7F88
0x00 0x00 0x01 0x31 | 0xA8 0x0F 0x00 0x02	1 - true	0x01 0x31 >> 1 = 152_{10}	2_{10}	0xA80F
0x00 0x00 0x05 0x90 | 0xE3 0xE3 0x00 0x00	0 - false	0x05 0x90 >> 1 = 712_{10}	0	0xE3E3

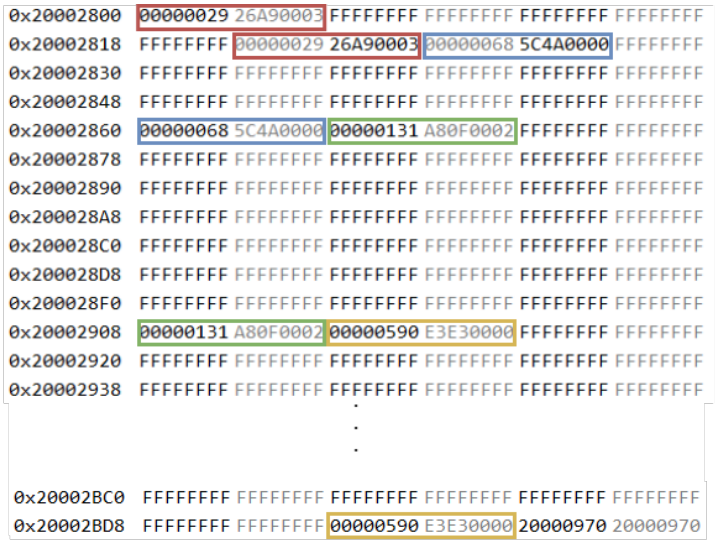
Figure 11: The heap after the Sheap allocator performed allocations for three memory blocks of the sizes 17, 50 and 150 bytes.

Let's assume that some work has been performed with the allocated memory and now the second memory block in figure 11 should be freed. In this case, a simple change of the allocation flag (A) to zero is enough, as the adjacent blocks are still allocated. The block is marked as not allocated and the allocator can reuse it if a request occurs and the block meets the requested size. Consider the same procedure, but with the third memory block instead of the second. When this block is freed, a simple change of the allocation flag will not suffice, as its successor is also not allocated. If the freeing of a memory block only adjusts the allocation flag, a bigger memory block cannot be restore even if all memory blocks have been freed. This means that although enough memory is theoretically available to fulfill a memory allocation request, the individual blocks on their own are too small to meet the required size. This is the reason the deallocation needs to perform coalescing. Please note that other approaches like *deferred coalescing* exist where the coalescing is not performed during the deallocation process. (Wilson et al. 1995, p. 22/43 [23]) The Sheap allocator focuses on simplicity and coalesces during the deallocation process.

Coalescing of Memory Blocks

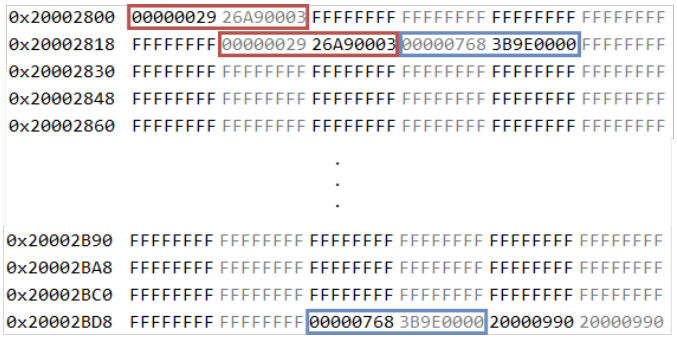
When a memory block is freed, the allocator needs to try to coalesce the freed block with the adjacent memory blocks. This is where the chosen memory layout, especially the boundary tags, comes in handy. A request to deallocate a resource needs to contain a reference to the resource. In the case of dynamic memory management, the user provides the memory address of the data to free. The address needs to be an address obtained by the allocator. As the caller is expected to provide a payload address (see figure 9 showing the allocation process), the Sheap allocator will subtract 8 bytes from the provided address to get the block header address. Using the information of the block header, the allocator can find the successor block and check if it is allocated or not. Furthermore, the allocator can easily check the predecessor block. If another 8 bytes are subtracted from the block header, the allocator can obtain the boundary tag of the predecessor block. The boundary tag contains the information if the predecessor is allocated or not. Due to the use of boundary tags, the check of the predecessor block is straightforward, and it can be performed in $\mathcal{O}(1)$. Without boundary tags, the determination of the predecessor will need a second traversal of the heap in this implementation. Depending on the adjacent blocks and their allocation flag, the current block is freed and, if possible, coalesced with its neighbours.

Figure 12 shows the heap memory after the second memory block (see figure 11) has been freed. As the adjacent blocks are not free, the only difference is the allocation flag (A). Figure 13 shows the heap after subsequently freeing the third memory block. This deallocation resulted in the coalescing of the third memory block with the second and fourth block. The current state of the heap now contains two memory blocks with sizes of 20 and 948 bytes. Taking into account the additional overhead per memory block of 16 bytes, we get the total heap size of 1000 $(948 + 20 + 2 * 16)$ bytes.



Memory block header	allocation flag	alignment size	alignment offset	crc
0x00 0x00 0x00 0x29 | 0x26 0xA9 0x00 0x03	1 - true	0x29 >> 1 = 20_{10}	3_{10}	0x26A9
0x00 0x00 0x00 0x68 | 0x7F 0x88 0x00 0x02	0 - false	0x68 >> 1 = 52_{10}	2_{10}	0x7F88
0x00 0x00 0x01 0x31 | 0xA8 0x0F 0x00 0x02	1 - true	0x01 0x31 >> 1 = 152_{10}	2_{10}	0xA80F
0x00 0x00 0x05 0x90 | 0xE3 0xE3 0x00 0x00	0 - false	0x05 0x90 >> 1 = 712_{10}	0	0xE3E3

Figure 12: The heap after the Sheap allocator performed allocations for three memory blocks of the sizes 17, 50 and 150 bytes and a subsequent deallocation for the second memory block.



Memory block header	allocation flag	alignment size	alignment offset	crc
0x00 0x00 0x00 0x29 | 0x26 0xA9 0x00 0x03	1 - true	0x29 >> 1 = 20_{10}	3_{10}	0x26A9
0x00 0x00 0x07 0x68 | 0x3B 0x9E 0x00 0x00	0 - false	0x07 0x68 >> 1 = 948_{10}	0_{10}	0x3B9E

Figure 13: The heap after the Sheap allocator performed allocations for three memory blocks of the sizes 17, 50 and 150 bytes and subsequent deallocation for the second memory block and following the third memory block.

4.1.2 Error Detection

The Sheap allocator provides additional safeguards to detect or even prevent errors related to dynamic memory management. The error checking is performed during allocation (malloc) or deallocation (free) operations. There is no distinct task in place which checks the heap periodically as of now. To provide information about an error to the user, the Sheaperd asserts are used.

The first error detection approach is to check if the address provided to the deallocation procedure is valid. The address itself is valid if it is not `NULL` and it is within the heap. If the address is invalid, an assert with appropriate error category will be executed. In the case of an invalid address, the error categories are `SHEAP_ERROR_NULL_FREE` or `SHEAP_ERROR_FREE_PTR_NOT_IN_HEAP`. (See listing 2)

If the supplied address is valid, the memory block header and the boundary tag are checked for validity. This check is performed using the crc checksum. When a memory block is created, the checksum is calculated using the first six bytes of the header and is written to the header and boundary of the memory block. The crc depends on the aligned size, the allocation flag (A) and the alignment offset. The crc is updated when a block is freed, as this changes the allocation flag (A). The memory block validation calculates the crc again and compares it with the crc stored in memory. If a mismatch occurs, an assert with the categories `SHEAP_ERROR_FREE_INVALID_HEADER` or `SHEAP_ERROR_FREE_INVALID_BOUNDARY` is executed. (See listing 2)

Since the previous check assured that the memory block header and boundary have valid crc checksums, it can be assumed they have not been altered. It is an assumption because we cannot restrict the memory access to the heap. By using pointer arithmetic and knowing the structure of the memory block, a user could intentionally adjust the memory header and boundary. Such malicious actions are not significant for the Sheap allocator. The allocator is intended to be used by developers for the purpose of error detection for their own software. With a valid memory block header, we can detect multiple deallocations, also known as double free. Multiple deallocation is a serious problem as it generally results in undefined behaviour[11]. The detection of such an error can be performed using the allocation flag (A) of the header. If the block is not allocated (the allocation flag (A) is zero), a multiple deallocation has occurred. In that case, an assert with the category `SHEAP_ERROR_DOUBLE_FREE` is executed. (See listing 2) The detection of multiple deallocation can help to mitigate *Use After Free* errors, which are a common source of problems and therefore ranked on place eight of the CWE Top 25 of 2020. (Mitre 2020, [13])

[11] Man page malloc(3) (accessed 26 May 2021):
https://man7.org/linux/man-pages/man3/free.3.html

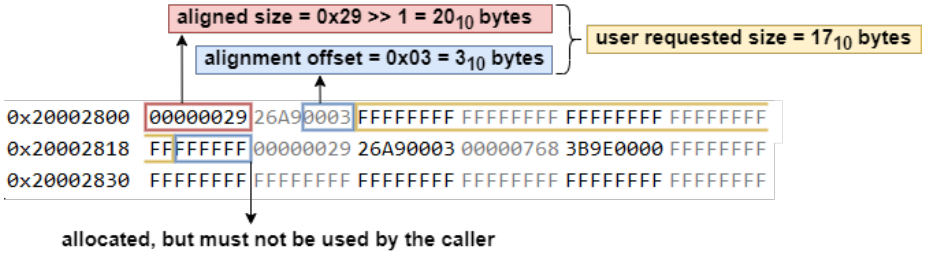


Figure 14: Aligned memory block in the heap with marked allocated user size and the additional alignment offset. A caller must not use the additionally aligned offset. The Sheap allocator can detect out of bounds writes directly behind the bound when freeing an aligned memory block if the data written out of the bound is distinguishable from the configurable `SHEAPERD_SHEAP_OVERWRITE_VALUE` (default: 0xFF).

As explained in 4.1.1, the memory block layout contains the aligned size as well as the alignment offset. That means that a memory block created during allocation can turn out to be larger than requested by the user. Consider the first memory block from the example heap in figure 11. Figure 14 shows the mentioned memory block divided into the requested size and the additional alignment. The user allocated 17 bytes of memory, which resulted in a 20 bytes memory block. The yellow region shows the 17 bytes of memory the user allocated. The two blue regions mark the size of the alignment offset stored in the header and the additionally allocated memory itself. The Sheap allocator can be configured to check the additionally allocated memory. If a define is available for `SHEAPERD_SHEAP_FREE_CHECK_UNALIGNED_SIZE`, the Sheap allocator will check the memory block for illegal writes in the additionally allocated memory. The define for `SHEAPERD_SHEAP_FREE_CHECK_UNALIGNED_SIZE` causes an additional define of `SHEAPERD_SHEAP_OVERWRITE_ON_FREE`. This define is used during the deallocation process. Freed memory will be overwritten with the `SHEAPERD_SHEAP_OVERWRITE_VALUE` (default: 0xFF). This enables the allocator to check if the aligned memory of a block only contains the `SHEAPERD_SHEAP_OVERWRITE_VALUE` value or something different. If something different is found, an out of bounds write has happened and an assert with the category `SHEAP_ERROR_OUT_OF_BOUND_WRITE` will be executed. (See listing 2)

Recording Allocations and Deallocations & Heap Statistics

The detection of an error is useful as it provides the certainty that something is going wrong. Finding the source of such a problem is an additional challenge. For example, if the Sheap allocator detects that a multiple deallocation has happened, it will execute the assert with the related category. The developer processing the callback gets the corresponding information and can try to locate the source of the error. Depending on the size and organisation of the project in use, the task of finding the possible free calls that could cause the problem

can be time and cost intensive. In the first approach to provide a mitigation for problems of this kind, the Sheap allocator provided additional macros for allocation and deallocation. These macros recorded the program counters of the callers. The general idea is to store some form of identification (e.g., program counter) during the allocation and deallocation process. This identification is stored in an array of user-defined size (`SHEAP_HEADER_ID_LOG_SIZE`). Furthermore, the identifications will be stored inside the memory block itself, if the extended memory block layout is used. These macros were able to record the program counter as they were directly inserted into the source code where they were used. Inline assembler was used to access the respective registers. The following listings (3 and 4) show the macros, including comments describing the individual lines of code. Both macros used the `r1` register to get hold of the program counter. Therefore, a backup of the current content of the `r1` register was needed. Following that, the content of the `pc` register was moved into `r1` and `r1` was subsequently used to move the content into a *C* variable (line 8 in listing 3). The variable was used in the following function calls to the `malloc` respectively the `free` function. After the function returned, the original content of the `r1` register was restored. The macros needed to temporarily use the `r1` register, because it is not possible to directly access the program counter from inline assembly. See the *ARM Compiler armcc User Guide* for detailed information. (ARM 2014, p. 282 [3])
As can be seen in the listings 3 and 4, the sheap allocation functions (`sheap_malloc` and `sheap_free`) have an additional parameter. It is also possible to call the functions directly and provide a user-defined form of identification instead of the program counter.

```
1  #define SHEAP_MALLOC(size, pVoid)                                    \
2  do {                                                                 \
3    if(ASSERT_TYPE(size_t, size)){                                     \
4      register uint32_t r1 asm("r1");                                  \
5      uint32_t r1Backup = r1;                                          \
6      __asm volatile("mov r1, pc\n");  /* Store the pc in r1     */ \
7      uint32_t pc;                                                     \
8      asm("mov %0, r1" : "=r" (pc));                                   \
9      size_t s = size;                                                 \
10     pVoid = sheap_malloc(size, pc);                                  \
11     r1 = r1Backup;                                                   \
12   }                                                                  \
13 } while(0)
```

Listing 3: Sheap malloc macro: The macro obtains the program counter from the register before calling the actual memory allocation. The `r1` register is used temporarily to get hold of the program counter and restored to its previous content after the memory allocation. The Sheap allocator uses the program counters to create a history of the allocation requests. Furthermore, the program counters can be used in the extended memory layout. (See chapter Extended Memory Block Layout)

```
#define SHEAP_FREE(pVoid)                                          \
do {                                                               \
  register uint32_t r1 asm("r1");                                  \
  uint32_t r1Backup = r1;                                          \
  __asm volatile("mov r1, pc\n");  /* Store the pc in r1       */ \
  uint32_t pc;                                                     \
  asm("mov %0, r1" : "=r" (pc));                                   \
  sheap_free(pVoid, pc);                                           \
  r1 = r1Backup;                                                   \
} while(0)
```

Listing 4: Sheap free macro: The macro obtains the program counter from the register before calling the actual memory deallocation. The `r1` register is used temporarily to get hold of the program counter and restored to its previous content after the memory deallocation. The Sheap allocator uses the program counters to create a history of the allocation requests. Furthermore, the program counters can be used in the extended memory layout. (See chapter Extended Memory Block Layout) .

The explained approach reached its limit when working with a different compiler. Chapter 5.1 presents the integration of the *Sheaperd* library into commercial applications. One of these applications uses the *Code Composer Studio (CCS)* development environment of *Texas Instruments (TI)*. This environment also contains the proprietary *TI Arm Clang Compiler Tools*. At the time of writing this thesis, the TI compiler tools did not support inline assembler, as it was used in the macros in the listings 3 and 4. Moreover, the usage of these macros in the first place originated from the idea of obtaining the current program counter. Using a function call for allocations and deallocations, as the libc implementation does, would change the program counter. Nevertheless, the goal was to provide an equivalent replacement for existing allocator calls. For this reason, the new approach is to use function calls to allocate and deallocate memory. To still be able to provide a form of identification, the link register (`lr`) is used. This is valid as a regular function call uses the `bl` (branch and link) instruction, which will copy the address of the next instruction into the `lr` register. (ARM Limited 2013, p. 157 [4]) This information provides the needed identification when tracking down errors. The first implementation attempt of this approach was in the form of a *C* function using inline assembler without accessing *C* variables. Despite the usage of the function attribute `naked`, which should instruct the compiler to not generate any prologue or epilogue code for this function, it inserted additional code nevertheless. This was further influenced by the compiler's optimisation settings. During debugging, the compiler optimisation may be disabled. The compiler inserted code disturbed the program execution. To overcome this obstacle, the naked *C* function was implemented entirely in assembler in a separate file. Using this approach, the compiler does not interfere with the assembler code. Unfortunately, different compilers may define different assembler directives. This means that the assembly file may need to be adjusted to contain specific assembler directives like for example `.func` and `.endfunc` for the gcc compiler or `.asmfunc` and `.endasmfunc` for the TI compiler. Listing 5

shows the assembly file for the gcc compiler. The assembly file initially pushes the used registers onto the stack. The link register (`lr`) is moved into the `r1` respectively `r2` register to be provided as a parameter to the subsequent function call. After the function returns, the previously pushed values are retracted (pop) from the stack to restore the original content and to return to the caller. Using this approach, the `lr` can automatically be used as identification and the adjustments that may be needed for the usage of a different compiler will only affect the assembly file.

```
1   .thumb
2   .global sheap_malloc_lr
3   .global sheap_calloc_lr
4   .global sheap_free_lr
5
6 sheap_malloc_lr:
7   .func
8
9   push        {r1, lr}
10  mov         r1, lr
11  bl          sheap_malloc
12  pop         {r1, pc}
13
14  .endfunc
15
16 sheap_calloc_lr:
17  .func
18
19  push        {r1, r2, lr}
20  mov         r2, lr
21  bl          sheap_calloc
22  pop         {r1, r2, pc}
23
24  .endfunc
25
26 sheap_free_lr:
27  .func
28
29  push        {r1, lr}
30  mov         r1, lr
31  bl sheap_free
32  pop         {r1, pc}
33
34  .endfunc
35
36  .end
```

Listing 5: The assembly file for the gcc compiler that provides the allocation and deallocations functions which automatically obtain the link register (*lr*) and use it as identification for the subsequent calls to the sheap allocator functions. The used registers are initially stored on the stack (push) and restored (pop) after the branch to the respective sheap function.

The recorded identifications can be obtained programmatically. If the identification is an address representing the caller, one can get the associated source file and line number using the `addr2line`[12] executable. For ease of use, the Sheaperd Eclipse plugin provides a view for this purpose. The user needs to provide the path to the `addr2line` executable and the binary of the project in use. With this information specified, the user can simply enter the corresponding address and obtain the source file and line number. Figure 15 shows an example of the plugin with an address obtained by the Sheap allocator.

Extended Memory Block Layout

The Sheap allocator provides an optional extended memory layout. If `SHEAPERD_SHEAP_USE_EXTENDED_HEADER` is defined as one, the allocator will use the extended memory layout. The extended layout expands the default layout with four additional bytes. The total increase of size (header and boundary) is eight bytes, resulting in a per block memory overhead of 24 bytes. The purpose of the additional layout is to provide more detailed information about allocations at runtime. The four additional bytes are used to store additional caller identification. Using this approach, each memory block that has been allocated or deallocated has an assigned identification. This information can be useful in different situations. For example, if a memory block is deallocated twice. If the identifications are stored as part of the memory block, the detection of which call to the allocator has already performed the deallocation can be performed in $\mathcal{O}(1)$. Figure 16 shows the heap after initialization and subsequent allocations of 17, 50 and 150 bytes. The identification is stored between the aligned size and the alignment offset (see figure 7) and it is included in the crc calculation. The last memory block has an identification value of 0×00000001. This is because this block has been created from the Sheap allocator and represents the remaining memory. The default identification value for automatic allocations can be defined using `SHEAPERD_SHEAP_AUTO_CREATED_BLOCK_ID` and defaults to one. The identification for a specific memory block can be obtained using the function `sheap_status_t sheap_getAllocationID(void* ptr, uint32_t* id)`. The source code file and line number can be obtained using the Sheaperd Eclipse plugin (see 15). This extended layout is optional to still provide an implementation with smaller memory overhead for restricted systems.

Heap statistics

The Sheap allocator provides a simple heap statistic. General heap information are stored and updated during the initialization, allocation and deallocation procedures. The statistic is available for the user by a function (`sheap_getHeapStatistic`). Refer to listing 6 to see which information is available and how to obtain the information.

[12] GNU Binutils addr2line (accessed 20 August 2021)
https://www.gnu.org/software/binutils/

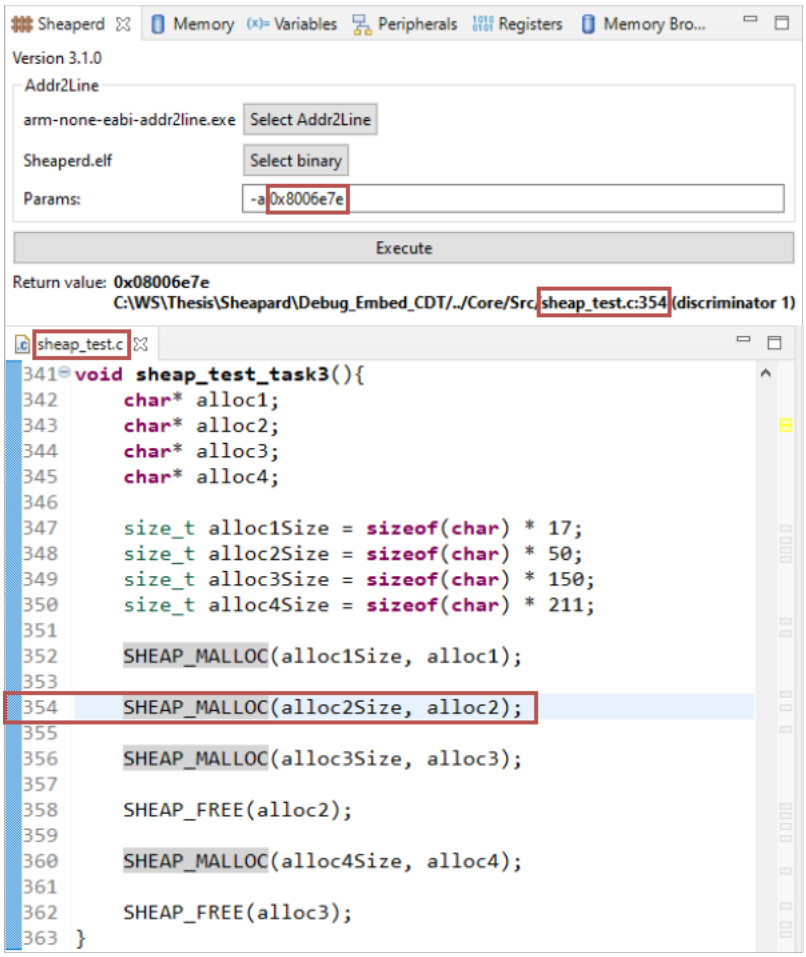


Figure 15: The Sheaperd Eclipse plugin provides a view which can be used to translate an address to the associated source file and line number using the **addr2line** executable.

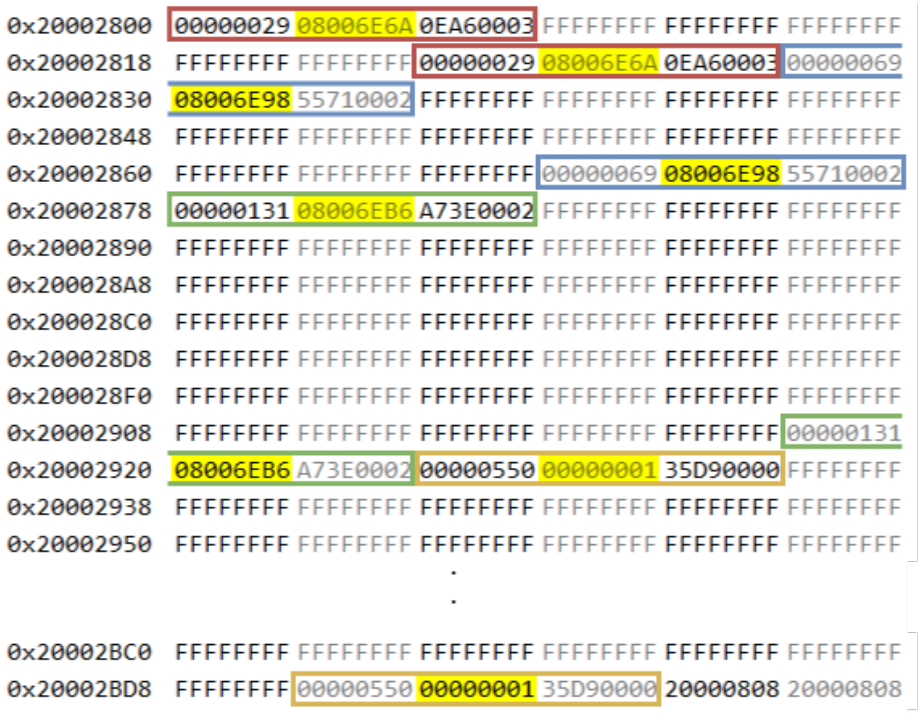


Figure 16: The heap after the Sheap allocator performed allocations, using the extended memory layout (Extended Memory Block Layout) for three memory blocks of the sizes 17, 50 and 150 bytes. The yellow highlighted parts of the headers/boundaries are the identifications of the caller that prompted the allocation.

```
typedef struct{
  uint8_t*  heapMin;
  uint8_t*  heapMax;
  uint32_t  currentAllocations;
  uint32_t  totalBytesAllocated;
  uint32_t  userDataAllocatedAlligned;
  uint32_t  userDataAllocated;
  size_t    size;
} sheap_heapStat_t;

void sheap_getHeapStatistic(sheap_heapStat_t* heapStat);
```

Listing 6: The Sheap heap statistic structure which is used to accumulate statistics of the current heap. It can be obtained by the user using the listed function.

4.2 Memory Protection Module & Stackguard

The *Memory Protection* of the *Sheaperd* library provides protection for user defined memory regions. Protecting memory regions can help to detect different types of errors. Consider a memory region which stores critical information. This information should only be altered using a specific function. In *C* the developer has full control of the whole memory and can access arbitrary memory locations (consciously and unconsciously). This means that although the critical information may only be altered consciously in a specific function, it is possible that the same data is altered in a different way unconsciously. If this data is crucial for the correct operation of the software, such an unnoticed change may lead to a system failure. The Memory Protection enables the developer to configure memory regions and protect them from unnoticed changes. Considering the example again, the protection can be disabled when the specific function needs to alter the information and enabled otherwise to protect the data from unwanted changes.
The Memory Protection is implemented using the *Memory Protection Unit (MPU)*. A detailed explanation of the MPU and the associated registers is available in chapter 4.2.1. The *Stackguard* part of the *Sheaperd* library provides an implementation that is using the Memory Protection to guard the stacks of different tasks. Embedded applications that use a RTOS can consist of multiple tasks. Each of these tasks has its own stack. Stackguard uses the Memory Protection and therefore the MPU to protect the stacks of individual tasks.

This chapter will give an overview of the Memory Protection and Stackguard implementation. The *Memory Protection Unit (MPU)* is the central component in regard to Memory Protection and is explained in Chapter 4.2.1. The current implementation is explained in chapters 4.2.2 and 4.2.3.

4.2.1 Memory Protection Unit

The *Memory Protection unit (MPU)* is an optional component included in different Cortex-M microprocessors (M0+, M3, M4, M7, M23, M33, M35P). As mentioned in the introduction, this thesis focuses on the Cortex-M3/4/7 microprocessors and therefore on the *ARMv7-M* architecture. Newer MPU features, introduced with the *ARMv8-M* architecture, are discussed in chapter 5.

MPU Registers

Using the MPU, one can define multiple memory regions. Each of these regions can be configured individually. The MPU provides five registers to configure the regions and their settings. Figure 17 shows the bit assignments of the MPU Type Register (`MPU_TYPE`).

MPU Type Register (`MPU_TYPE`)

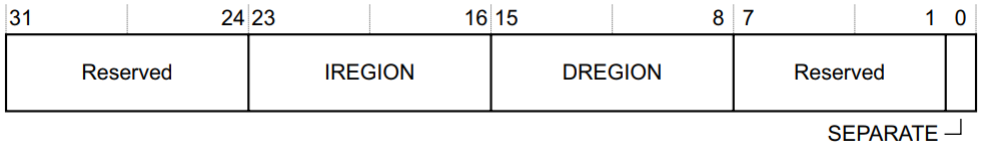


Figure 17: The MPU Type Register (`MPU_TYPE`) provides information about the number of available regions. The processor does not provide a MPU if the `DREGION` field is zero. (Source: ARM 2010, p. 636 [5])

The `DREGION` field specifies how many MPU regions are available. The fields `IREGION` and `SEPARATE` are not of significance, as the *ARMv7-M* architecture only supports a unified MPU. A unified MPU means that there is no distinction between instruction regions (`IREGION`) and data regions (`DREGION`). The `MPU_TYPE` register is available even if the MPU is not implemented by the Cortex-M device in use. The `DREGION` field holds the number of available memory regions. The MPU is not implemented if this field reads zero. (ARM 2010, p. 636 [5])

MPU Control Register (`MPU_CTRL`)

The activation of the MPU, the memory background region and the interrupt MPU activation is configured using the MPU Control Register (`MPU_CTRL`). Figure 18 shows bit assignments of the MPU Control Register. The `ENABLE` bit is used to enable and disable the MPU. The `HFNMIENA` bit controls if the MPU

is active when handling interrupts with a priority less than zero. The Memory Protection implementation sets this bit to zero to not restrict user implemented fault handlers. The `PRIVDEFENA` bit controls if the default memory map is accessible when running in privileged mode. If this bit is not set, any memory access which is not explicitly allowed in the MPU configuration will result in a `MEMFAULT`. (ARM 2010, p. 637 f. [5])

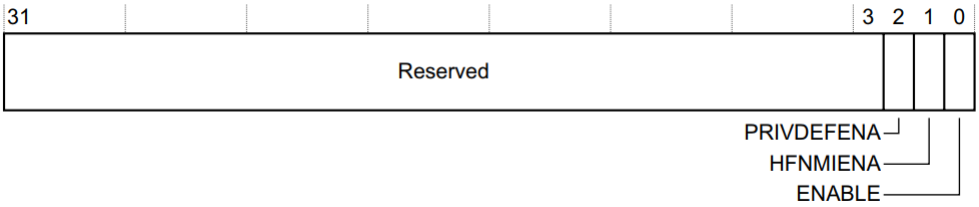


Figure 18: The MPU Control Register (`MPU_CTRL`) is used to enable and disable the MPU, the memory background region and the hard fault behaviour. (Source: ARM 2010, p. 637 [5])

MPU Region Number Register (`MPU_RNR`)

The region selection can be performed using the MPU Region Number Register (`MPU_RNR`). Figure 19 shows bit assignments of the MPU Region Number Register. The `REGION` field is used to select the currently active region. Settings specified in the following registers(`MPU_RBAR` and `MPU_RASR`) will take effect upon the selected region in the `REGION` field. The region selection can be performed in an alternative way, which reduces the number of register accesses to configure a region by one. (See MPU_RBAR) (ARM 2010, p. 638 f. [5])

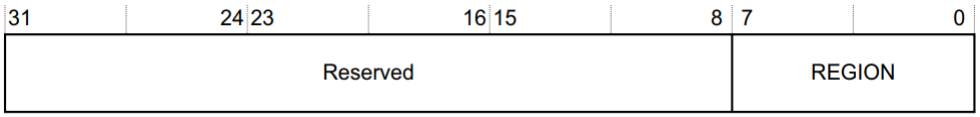


Figure 19: The MPU Region Number Register (`MPU_RNR`) is used to select the currently active region. Adjustments in the `MPU_RBAR` and `MPU_RASR` registers will affect the region selected in the `REGION` field. (Source: ARM 2010, p. 638 [5])

MPU Region Base Address Register (`MPU_RBAR`)

The MPU Region Base Address Register (`MPU_RBAR`) is the first register to configure settings for a specific region. Figure 20 shows the bit assignments of the

MPU Region Base Address Register. The base address of the current region is defined using the `ADDR` field. The `VALID` bit can be used to select a region and set the base address with a single register access. Alternatively, the region can be specified with separate access to the `MPU_RNR` register. If the `VALID` bit is set, the `REGION` field in the `MPU_RNR` register is updated to the value specified in the `REGION` field automatically.
Figure 20 shows that the `ADDR` field consists of only 27 bits. This means that not any address can be used as base address. The minimum supported alignment of the base address is implementation defined. The ST board used for testing (STM32L476G-DISCO) has a minimum region alignment of 32 bytes. Therefore, the smallest possible region size is 32 bytes. **It is important to note that the base address needs to be aligned to the size specified in the `MPU_RASR` register at all times.** If a region size of 512 bytes is configured, the `ADDR` field in the `MPU_RBAR` register needs to be aligned to a multiple of 512 bytes, for example, `0x20002800` or `0x20002A00`. The developer is responsible to make sure that the region base address is properly aligned in regard to the configured size. (ARM 2010, p. 639 [5])

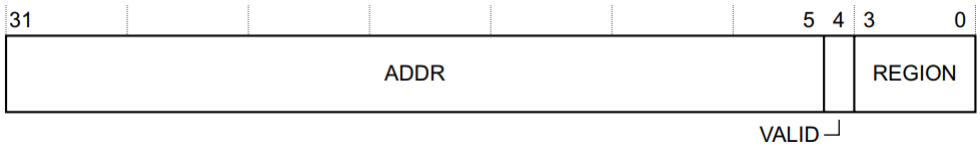


Figure 20: The MPU Region Base Address Register (`MPU_RBAR`) is used to specify the base address of the selected region. The address can be specified in the `ADDR` field. The `REGION` field can be used in combination with the `VALID` field to update the region in the `MPU_RNR` register. (Source: ARM 2010, p. 639 [5])

MPU Region Attribute and Size Register (`MPU_RASR`)

The main part of the region configuration is done using the MPU Region Attribute and Size Register (`MPU_RASR`). Figure 21 shows bit assignments of the MPU Region Attribute and Size Register.

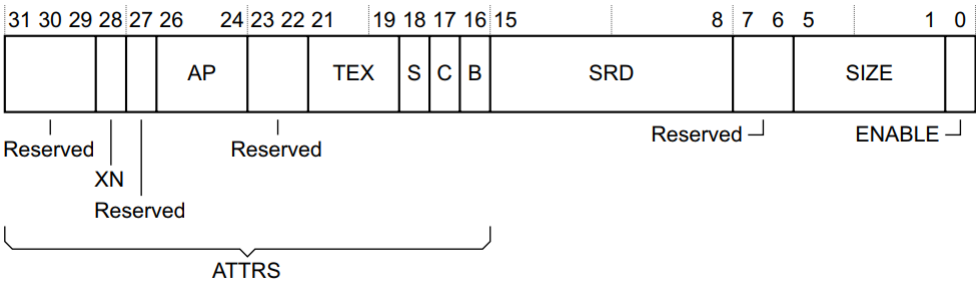


Figure 21: MPU Region Attribute and Size Register (`MPU_RASR`) is used to configure the size, the activation, the subregions and the attributes of the current region. (Source: ARM 2010, p. 640 [5])

The `ENABLE` bit is used to enable and disable the current region. The current region is identified using the `MPU_RNR` register. Using the `SIZE` field, one can specify the size of the current region. The `SIZE` field consists of five bits and is interpreted as an exponent. The content of the `SIZE` field is increased by one and interpreted as the exponent(power) of the base two (1).

$$regionSize[byte] := 2^{(\texttt{SIZE}+1)} \tag{1}$$

As the minimum supported region size is 32 bytes, the smallest valid value for the `SIZE` field is four (2).

$$regionSize[byte] := 2^{(4+1)} = 2^5 = 32 \tag{2}$$

The `SRD` (Subregion Disable) field controls the activation of the subregions of the current region. MPU regions with a size of at least 256 bytes are automatically subdivided by the MPU into eight subregions of equal size. Each bit in the `SRD` field controls if the associated subregion is enabled (0) or disabled (1). The least significant bit in the `SRD` field represents the subregions with the lowest address, and the most significant bit represents the highest address. If the current region is smaller than 256 bytes and the bits of the `SRD` field are set, the effect is unpredictable. (Source: ARM 2010, p. 640 f. [5])

The `ATTRS` field is subdivided into multiple fields. We start of with the `TEX` (type extension) field as well as the `S` (shareable), `C` (cacheable) and `B` (bufferable) bits. These settings are used to configure the type of memory respectively the type of device (memory-mapped I/O) of the current region. Furthermore, These settings are exported to the bus system together with each data access. (Source: Yiu 2014, p. 361 [24]) Figure 22 shows how the system makes use of the memory access attributes. This configuration in general depends on the implementation and therefore one needs to check the vendor documentation of the specific device in use. For example, *STMicroelectronics* states that the

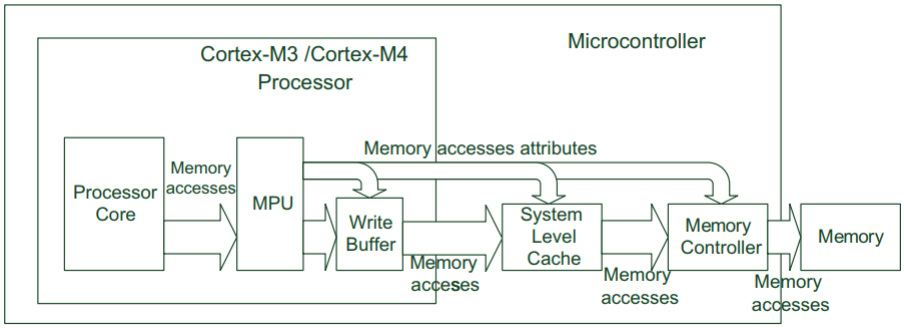


Figure 22: The ARM Cortex-M3/M4 can use the MPU memory attributes internally and additionally propagates them to the external system. The external system can make use of the attributes as well. For example, an external cache can check the `C` (cacheable) flag to see if caching is permitted. (Source: Yiu 2014, p. 363 [24])

Memory region	TEX	C	B	S	Memory type and attributes
Flash memory	b000	1	0	0	Normal memory, Non-shareable, write-through
Internal SRAM	b000	1	0	1	Normal memory, Shareable, write-through
External SRAM	b000	1	1	1	Normal memory, Shareable, write-back, write-allocate
Peripherals	b000	0	1	1	Device memory, Shareable

Figure 23: STMicroelectronics recommendation for the configuration of the MPU region attributes for their Cortex-M4 implementation. (Source: STM 2020, p. 199 [17])

shareable and cacheable attributes **do not** affect the system behaviour of their Cortex-M4 implementation. Nevertheless, they advise to properly configure the register to keep the application code portable. STMicroelectronics also provides a recommendation for the configuration of the memory region attributes, which is depicted in figure 23. Note that in STM implementations, the memory access using Direct memory access (DMA) **is not affected by the MPU attributes**. To control DMA access, the MPU needs to control the DMA registers of the Microcontroller unit (MCU). (STM 2020, p. 199 [17])

The `AP` (access permission) field defines in which way the current memory region can be accessed. Figure 24 shows the possible values for the `AP` field. The access permissions distinguish between privileged and unprivileged access. The privileged and unprivileged mode are the two software execution levels in the *ARMv7-M* architecture. (ARM 2010, p. 512 [5])

The `XN` (execute never) bit controls if it is possible to execute code from the current region. A value of zero indicates that the execution of instructions

AP[2:0]	Privileged access	Unprivileged access	Notes
000	No access	No access	Any access generates a permission fault
001	Read/write	No access	Privileged access only
010	Read/write	Read-only	Any unprivileged write generates a permission fault
011	Read/write	Read/write	Full access
100	UNPREDICTABLE	UNPREDICTABLE	Reserved
101	Read-only	No access	Privileged read-only
110	Read-only	Read-only	Privileged and unprivileged read-only
111	Read-only	Read-only	Privileged and unprivileged read-only

Figure 24: The possible `AP` field values of the `MPU_RASR` register. The effectively resulting permission can be obtained from the access and the note columns. (Source: ARM 2010, p. 642 [5])

fetched from the current region is permitted. A value of one prevents the execution of an instruction fetched from the current region. Additionally, the read access in the `AP` field needs to be configured for the current execution privilege mode to be able to execute a fetched instruction. If the read access is not granted, or the `XN` bit is set to one, the processor will generate a MemFault when the instruction is issued for execution. (Source: ARM 2010, p. 642 [5])

To provide a complete description of the MPU registers, the `MPU_RBAR` and `MPU_RASR` alias registers need to be mentioned. Using these register aliases along with the alternative region selection, mentioned in MPU_RBAR, software can use a stream of word writes to efficiently update up to four MPU regions. The registers are located at offsets of 8, 16, and 24 bytes from the `MPU_RBAR` address of `0xE000ED9C` (Source: ARM 2010, p. 642 [5])

4.2.2 Memory Protection Module

The Memory Protection Module provides the base functionality to configure MPU regions. It directly interacts with the MPU registers mentioned in MPU Registers. The interface of the module provides an abstraction to the user in the form of the `mpu_region_t` typedef available in listing 7. The members `size` and `ap` are enumerations which only provide valid values for the associated register fields `SIZE` and `AP`. (See MPU Region Attribute and Size Register (`MPU_RASR`)) The module also provides function to enable or disable the MPU. (`memory_protection_enableMPU()` and `memory_protection_disableMPU()`) The module automatically disables the MPU when configuring a new region and enables it afterwards. Moreover, the module makes sure that the necessary

memory barrier instructions, which are needed to make sure that the changes to the MPU propagate properly (see STM 2020, p. 196 f. [17]), are performed. Additionally, the Memory Protection Module checks if the provided region to configure is valid. The base address of the region is verified to be at least aligned to 32 bytes. The base address is furthermore checked if it aligns properly according to the specified size of the region. The module also makes sure that the provided region number is valid. If any invalid setting is found, an error will be returned to the caller. The Memory Protection Module is furthermore used by Stackguard to configure the regions according to the stacks of individual tasks. The module has intentionally been developed independently of Stackguard, so it can be reused for additional use cases for the MPU besides stack isolation.

```
typedef struct {
    uint32_t address;
    bool enabled;
    uint8_t number;
    uint8_t srd;
    mpu_regionSize_t size;
    mpu_access_permission_t ap;
    bool cachable;
    bool bufferable;
    bool shareable;
    uint8_t tex;
    bool xn;
} mpu_region_t;
```

Listing 7: The `mpu_region_t` typedef, which used to configure MPU memory regions using the Memory Protection Module.

4.2.3 Stackguard

The Stackguard part of the Sheaperd library is intended to be used to protect the stacks of different tasks in an application using a RTOS. Embedded applications that make use of a RTOS can consist of multiple tasks. Each of these tasks will have its own stack. It depends on the RTOS in use, where in memory the stack areas reside. In most cases, there is a designated area where the stacks of the different tasks are stored. The stack is a private area of memory which should only be accessed by the associated task. A task accessing the stack of another task is usually the result of an error. Stackguard prevents such errors, using the Memory Protection Module to set up MPU region for the stacks.

Stackguard Configuration

Stackguard is configured using four functions. The function `stackguard_init( stackguard_memFault_cb memFaultCallback)` initializes the internals and checks if the MPU is available on the current device. The user can provide a callback

to Stackguard which is called if a MemFault with a data access violation occurs. A task can be added using the function `stackguard_addTask(uint32_t taskId, uint32_t* sp, mpu_regionSize_t stackSize)`. The stack pointer (`sp`) is used as base address for a MPU region of size `stackSize`. As Stackguard is using the Memory Protection Module, the restrictions on the base address alignment and the size also apply here. This complicates the creation of tasks using a RTOS. The user needs to make sure that the stacks are properly aligned. How to enforce the alignment depends on the RTOS in use. *FreeRTOS*[13], for example, enables the user to create a task with a user-provided stack. The user can align the stack properly and create a task using the prepared stack. There is also the possibility to use a define (`portBYTE_ALIGNMENT`) to influence the stack alignment. After adding a task, a corresponding MPU region with access permission **"no access"** exists. Tasks can be removed from using the function `stackguard_removeTask(uint32_t taskId)`. The function `stackguard_guard()` enables the MPU using the Memory Protection Module.

Stackguard Execution

Stackguard needs to know what task is currently executing. Depending on this information, it can adjust the access permissions (see figure 24) for the MPU regions accordingly. The RTOS in use is responsible for task switching and knows which task is executing. How to obtain this information depends again on the RTOS in use. FreeRTOS, for example, provides macros for the task life cycle. The user can provide implementations for the macros `traceTASK_SWITCHED_IN` and `traceTASK_SWITCHED_OUT` and FreeRTOS will call these macros when performing a task switch. This information can be used to let Stackguard know which task is executing. Therefore, Stackguard provides the function `stackguard_taskSwitchIn(uint32_t taskId)`. When calling this function, Stackguard will loop through all added regions and adjust the access permissions. The region of the switched-in task will have full access and for all other regions, the access will be denied.

If access to a memory address which lies within an MPU region is performed and the access permission of the region is not met, a MemFault will be triggered. Stackguard can provide a handler for MemFault exceptions. If the `STACKGUARD_USE_MEMFAULT_HANDLER` define is set to one, the MemFault handler will be available. The Stackguard MemFault handler will check if the fault occurred because of a data access violation. If that is the case, the user-provided callback will be called with the fault address as well as the exception stack frame. Cortex-M devices using the *ARMv7-M* architecture automatically push information onto the stack when entering an exception. Listing 8 depicts the so-called basic frame. If a Floating Point Unit (FPU) is implemented, the exception entry behaviour may push an extended frame onto the stack. (ARM 2010, pp. 535-539 [5]) Howsoever, the extended frame enhances the basic frame

[13] FreeRTOS (accessed 23 June 2021): https://www.freertos.org/

and Stackguard does not need to distinguish between these two. When receiving the callback, the user has the information at which address the violation happened (`faultAddress`). Furthermore, the user can make use of the basic frame and, for example, can obtain the `return_address` which indicates where the data access violation occurred.

Information about occurring exceptions can generally be obtained from the Configurable Fault Status Register (`CFSR`). The `CFSR` consist of the MemManage Status Register (`MMFSR`), the BusFault Status Register (`BFSR`) and the UsageFault Status Register (`UFSR`). Detailed information is available in the *ARMv7-M* architecture reference manual. (ARM 2010, p. 609-612 [5])

```
#pragma pack(1)
typedef struct {
    uint32_t r0;
    uint32_t r1;
    uint32_t r2;
    uint32_t r3;
    uint32_t r12;
    uint32_t lr;
    uint32_t return_address;
    uint32_t xpsr;
} stackguard_stackFrame_t;
#pragma pack()
```

Listing 8: The stack frame typedef, which represents the basic stack frame which is automatically pushed onto the stack as part of the exception entry behaviour of a Cortex-M device using the *ARMv7-M* architecture. (ARM 2010, p. 536 [5])

4.3 Array Bound Asserter

To be able to create asserts for arrays in the *C* programming language, first the source code needs to be parsed. Different *C* parsers are available, with one of them being the Eclipse C/C++ Development Tooling (CDT) implementation. The CDT provides an AST of source code files which can be traversed and augmented with additional statement, like assertions. Due to the fact that the Eclipse IDE is freely available, widely used and provides tooling to develop plugins which can access the available features like the CDT, the array bound asserter was implemented as a Eclipse plugin. The array bound asserter is implemented as part of the Sheaperd Eclipse plugin. The plugin has already been mentioned in section 4.1.2 with reference to the executor view. The array bound asserter is intended to support the developer during debugging and testing. The *C* programming language contains no concept of boundary checks for arrays. If a measure of this kind is desired, it needs to be performed manually. The seriousness of this kind of errors is evident if you consider the CWE top 25 of 2020. Out of bound errors reside on place two (*Out-of-bounds Write*) respectively place four (*Out-of-bounds Read*) in the ranking. (Mitre 2020, [13]) The Sheaperd array asserter is part of the Sheaperd Eclipse plugin and provides array assertions for the *C* programming language.

This chapter will give an overview of the array asserter implementation. The Eclipse plugin based on Eclipse C/C++ Development Tooling (CDT) and the available AST is described in chapter 4.3.1. Chapter 4.3.2 describes the modification of the AST to generate the specific assertions.

4.3.1 Eclipse Plugin and CDT Integration

Eclipse provides a standalone IDE called Plugin Development Environment (PDE). The PDE provides tools to develop, debug and deploy Eclipse plugins. The Sheaperd Eclipse plugin has been created using this IDE. The PDE provides basic building blocks to enhance the usual Eclipse UI with additional view components. The executor view has already been mentioned in 4.1.2. The array assertion is another part of the plugin. It is accessible through the context menu on Eclipse C-Projects. The assertion can be performed on single files or on folders/projects containing multiple files. The execution of the assertion is logged using the console view and provides links to the inserted assertions. The context menu also provides an entry to undo the assertions.

The *C* source code files are parsed using the Eclipse CDT. The CDT provides APIs to obtain the translation units of the files. Using these translation units, the associated AST can be created. The Sheaperd Eclipse plugin uses the AST created from the CDT to insert the array assertions. The AST can be accessed by creating an implementation of the abstract `ASTVisitor` class. As the name suggests, the `ASTVisitor` follows the visitor pattern and provides a `visit` and a `leave` method. These methods are overloaded several times for the different kinds of nodes of the AST. Furthermore, the CDT provides a rewriter for the

AST. Changes to the AST can be accumulated during the traversal and executed afterwards. The changes will be reflected in the *C* source code file.

4.3.2 Modifying the Abstract Syntax Tree (AST)

To create assertions for arrays in a *C* source file, the corresponding nodes in the AST need to be found. Additionally, the size of the array is needed. The array size is obtained using the overloaded `leave` method for array declaration. This method is called when an array declaration statement has been traversed. At this point, information about the array containing the name and the size is stored. Local and global arrays have their own storage location. The local storage is cleared when the AST traversal leaves a function.
To find an array access statement inside the AST the overloaded `visit` method for statements is used. A statement in this context is generic and the plugin checks if any of the different statements (if statement, while statement, for statement, expression statement, etc.) contains an array subscript expression in any form. If a subscript expression is found, the associated statement is used as a reference point for the insertion of the assertion. To create a correct assertion, the index of the array access needs to be obtained. In earlier versions of the plugin, the index of the array access has directly been used for the assertion. This turned out to be problematic if the subscript contained pre- or post increments/decrements. Listing 9 shows an assertion from an earlier version of the plugin and demonstrates this problem. The array subscript `++i` from line five is directly used in the assertion in line four. As this statement changes, the value of `i` the assertion will interfere with the program execution and the array access in line five would be an out-of-bounds write.

```
1  void foo() {
2      int values[2] = { 0 };
3      int i = 0;
4      int j = values[++i];
5  }
6
7  void foo() {
8      int values[2] = { 0 };
9      int i = 0;
10     SHEAPERD_ASSERT("Plugin: Array bound check failed.", ++i < 2,
           SHEAPERD_ARRAY_BOUND_CHECK);
11     int j = values[++i];
12 }
```

Listing 9: The original function as well as the erroneous assertion from earlier versions of the Sheaperd Eclipse plugin resulting from directly using the array subscript as assertion condition.

The current version of the plugin (V3.5.0) checks if the array subscript contains any pre- or postfix increment/decrement operators and adjusts the assertion condition accordingly. Listing 10 shows how the problem shown in listing 9

is handled in the current version. Besides inserting the assertions themself, the plugin also makes sure that the needed header file (`sheaperd.h`) is included.

```
1 void foo() {
2     int values[2] = { 0 };
3     int i = 0;
4     SHEAPERD_ASSERT("Plugin: Array bound check failed.",
            i + 1 < 2, SHEAPERD_ARRAY_BOUND_CHECK);
5     int j = values[++i];
6 }
```

Listing 10: The current Sheaperd Eclipse plugin (V3.5.0) checks the array subscript for pre- or postfix increment/decrement operators (line five) and adjusts the assertion condition accordingly (line four).

During the assertion process, the integrated Eclipse console is used to record the progress. The plugin records which files have been found, which file is currently being traversed, and the inserted assertions. The assertions are recorded as hyperlinks which can directly be used to open the specific insertion location. To remove the assertions the plugin traverses the AST and checks if any plugin created assertion statement can be found. If a node is found, it can directly be removed from the AST. The removal is also recorded in the Eclipse console. Figure 25 shows the console output from the assertion on a project folder.

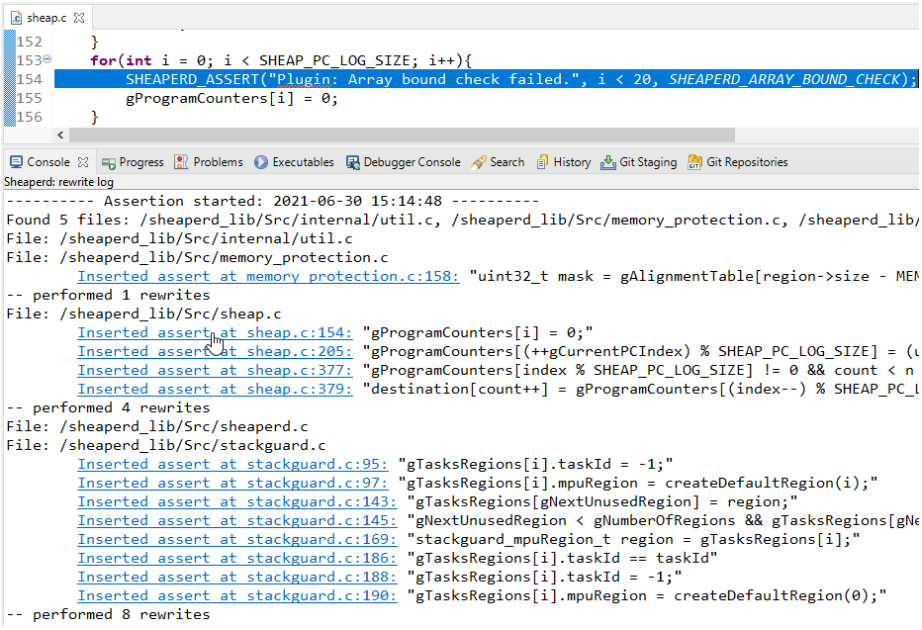


Figure 25: The Eclipse console of the Sheaperd Eclipse plugin. The console is used to record the inserted assertions. The hyperlinks can be used to directly open the insertion location.

5 Results and Discussion

This chapter provides a conclusion about the implementation and concrete usage of the Sheaperd library. The initial chapter 5.1 describes the integration of the Sheaperd library into existing commercial embedded software applications. The reader can develop understanding on how to approach the task of implementing the library into existing applications. The chapter elaborates on specific pitfalls and how to prevent or mitigate them. Chapter 5.2 discusses the current implementation of the Sheaperd library, including specific design decisions. Furthermore, it provides an outlook on how the Sheaperd library can be further developed and extended. The final chapter 5.3 introduces features of the *ARMv8-M* architecture and explains how these could be used in future versions of the Sheaperd library.

5.1 Case Study - Integrating Sheaperd into Commercial Applications

This chapter verifies that the Sheaperd library can be integrated into existing embedded software applications. The integration is described from Stackguards perspective in the chapters 5.1.1 and 5.1.2. The integration of the Sheap allocator and the array assertion is described in chapter 5.1.3. Chapter 5.1.4 describes the execution of the applications, including the emerging faults and how they are handled. The closing Chapter 5.1.5 provides a conclusion of the integration.

5.1.1 Stack Alignment

In this case study, the Stackguard part of the Sheaperd library is added to commercial embedded software applications. The purpose of Stackguard is to protect the stacks of different tasks executing in an RTOS. One application being studied is using the *Keil RTX* RTOS together with the Common Microcontroller Software Interface Standard (CMSIS) RTOS API v1. In this thesis, the term task will be used synonymously to the term thread as used in the CMSIS API. As mentioned in 4.2.3, the stack of a task needs to be properly aligned depending on the size of the stack. The CMSIS v1 does not provide the possibility to create a task with a manually allocated stack. It is possible to specify the size of the stack, but it will be allocated automatically. Because of this limitation, the task creation needed to be adjusted to guarantee that the stacks are properly aligned according to their size. For reasons of simplicity, to keep the changes to the application code at a minimum and to not alter the CMSIS v1 API this adjustment was done manually. The task creation function of the *Keil RTX* was altered to provide the correctly aligned stacks to the associated tasks. Other RTOS APIs commonly provide possibilities to create tasks with user allocated stacks. After the task has been created, the *Keil RTX* RTOS executes a notify macro. This is used to realize when a new task has been created. Each newly created task is added to Stackguard.

Another application under consideration uses the *Texas Instruments (TI)* RTOS. The task stack can be provided manually to the task creation process and therefore the alignment needed for the MPU (see 4.2.3) can be ensured. A so-called *HookSet* provides callbacks for the task creation and switching process.

5.1.2 Context Switching

Once the correct stack alignment has been ensured, the next step is to notice when a context switch occurs. The *Keil RTX* RTOS provides a notify macro which is called when a task switch occurred. The notify macro was used to call the `stackguard_taskSwitchIn(uint32_t taskId)` function to inform Stackguard that a task switch has occurred and the MPU regions need to be adjusted. After building and starting the application, virtually immediately a memory fault occurred. The reason for this fault is that the task-switch macro is called before the task switch has been performed completely. The fault occurred inside the SVC handler. Figure 26 shows the SVC handler of the *Keil RTX* which is implemented in assembler. The memory fault occurred at line 194 (2.). The instruction in line 194 is the store multiple (STM) instruction. This instruction takes a base address (R12) and, separated by a comma, multiple registers (R0-R2) specified in curly brackets. The registers inside the curly brackets are stored at the base address. In this specific case, the base address (R12) represents the process stack pointer (PSP). Line 193 stores the PSP into R12. The memory fault occurred on line 194 because of the branch to a *C* function in line 191 (1.). During the execution of this function, eventually the task-switch notify macro is called. This is the task switch macro that has been used to update the MPU regions. Access to the stack of the switched in task is enabled, and access to all other stacks is disabled. When the code execution returns from the branch in line 191 (1.) it will subsequently reach line 194 and tries to write to the stack of a task that has already been switched out from Stackguard's perspective. This results in a memory fault, as the access to the stack has already been disabled in the respective MPU region.

To overcome this limitation, the notify macro needs to be executed after all the necessary stack accesses have been performed. The SVC handler performs additional code if a task switch should be performed. Line 199 exits if no task switch is performed in this SVC. In this case, no notify macro must be called. If a task switch is performed, the code execution continues as additional handling is needed. In line 206 (3.), the handler branches to a *C* function that checks if a stack overflow occurred. At the end of this function, an additional task notify macro has been added manually, and the original macro has been deactivated. As this macro is now executed after all necessary accesses to the switched-out stack have been performed, the Stackguard protection is now configured properly and will guard the individual task stacks.

The application using the *TI* RTOS behaved similarly. As mentioned, the task switching callback can be defined using a HookSet. When this is called from

```
180 SVC_Handler_Veneer:
181         .endif
182         .fnstart
183         .cantunwind
184
185         MRS     R0,PSP                  /* Read PSP */
186         LDR     R1,[R0,#24]             /* Read Saved PC from Stack */
187         LDRB    R1,[R1,#-2]             /* Load SVC Number */
188         CBNZ    R1,SVC_User
189
190         LDM     R0,{R0-R3,R12}          /* Read R0-R3,R12 from stack */
191         BLX     R12                     /* Call SVC Function */     1.
192
193         MRS     R12,PSP                 /* Read PSP */
194         STM     R12,{R0-R2}             /* Store return values */   2.
195
196         LDR     R3,=os_tsk
197         LDM     R3,{R1,R2}              /* os_tsk.run, os_tsk.next */
198         CMP     R1,R2
199         BEQ     SVC_Exit                /* no task switch */
200
201         CBZ     R1,SVC_Next             /* Runtask deleted? */
202         STMDB   R12!,{R4-R11}           /* Save Old context */
203         STR     R12,[R1,#TCB_TSTACK]    /* Update os_tsk.run->tsk_stack */
204
205         PUSH    {R2,R3}
206         BL      rt_stk_check            /* Check for Stack overflow */  3.
207         POP     {R2,R3}
208
```

Figure 26: The *Keil RTX* SCV handler is implemented in assembler. Line 191 (1.) branches to *C* functions that handle the specific SVCs. Line 194 (2.) stores return values to the current stack. In line 206 (3.) a stack overflow check is performed if a task switch occurred during the SVC.

the RTOS and the MPU is configured and enabled, a memory fault occurs. As in the case of the *Keil RTX* RTOS, the *TI* RTOS also needs to access the stack of the switched out task after the callback. This problem was solved by not directly enabling the MPU after the task switch, but delay the activation until the `Task_enter` function is called.

5.1.3 Sheap and Array Assertion

The first application under consideration does not use a lot of dynamic memory allocation. The three tasks use three allocations in total, which have been rewritten manually to use the Sheap allocation, respectively deallocation macros. (See 4.1.2) The heap itself was initialized using information from the linker script. The linker script defines the symbol `_end`, which is used to obtain the start address of the heap. For the heap size, the minimum defined heap size of `0x400` (1024_{10}) bytes is used. The array assertion has been performed on all source code files of the custom application code. The assertion callback is used to observe if any assertion or dynamic memory allocation/deallocation failed. The array assertions as well as the Sheap allocator did not report any unmet assertions. The library code of the MCU manufacturer has not been edited in regard to array assertion or dynamic memory allocation.

The second application under consideration contains more dynamic memory allocations. Furthermore, the first approach to perform allocations with automatically obtained identification using macros failed for this application. A new approach was implemented in the form of functions implemented in plain assembler, which can also automatically obtain identification for the allocations. (See chapter 4.1.2 for detailed information) As this application used significantly more dynamic memory allocations as the first application, the manual replacement of the allocations was tedious. An automatic replacement could be an additional feature of the Sheaperd Eclipse plugin (see 5.2). The Sheap allocator reported free attempts of *NULL* values, which is generally not an error itself if the pointer has not been freed unintentionally.

5.1.4 Executing the Application

As that Stackguard is properly integrated into the *Keil RTX* RTOS of the first application (See 5.1.2), it can be launched and the stacks will be guarded against access from other tasks except the stack owner. A first stack related memory fault occurred after creating the application task from within the main task. The *Keil RTX* RTOS can be configured to wrap the main function inside a task. This feature is used for the application in consideration. From the main task, two additional tasks are created. One of them, the application task, is initialized with arguments (`void* arguments`). The passed argument's pointer is obtained from a local variable of the main task. The arguments' pointer is passed through the *Keil RTX* task creation and ends up as a parameter to the application task function. The application task casts the `void*` to the actual

data type and updates its value. This results in a memory fault. As mentioned, the arguments' pointer is obtained from a local variable of the main task. The local variables are stored on the stack. This means that the passed pointer references a variable which is allocated on the main stack. Write access to this variable from another task as the main task, like the application task in this case, is not allowed and correctly results in a memory fault. This problem can be solved by moving the arguments variable of the main thread from the local scope into the global scope. In *C* global variables are not stored on the stack, but depending on their initialization in the data segment or the bss segment. If a global variable has a value other than zero, it will reside in the data segment, otherwise they reside in the bss segment. The whole bss segment is initialized with zero. (Erickson 2008, p. 75 [8]) After moving the arguments variable to global scope, the application task could access it without any memory access violation.

Another stack-related memory fault occurred inside the *Keil RTX* RTOS. Some functions of the RTOS perform direct access to the stack of a task not currently running. For example, the function `rt_mbx_send(OS_ID mailbox, void* p_msg, U16 timeout)` proofed to be problematic. As the name suggests, the function tries to send a message to a specific mailbox. A special case in the message delivery occurs when a task is already waiting for a message to be delivered. In this case, the RTOS will not put the message into a mailbox, but it will directly deliver it to the waiting task. This delivery is performed by writing the message directly to the stack of the waiting task. As this task is currently not executing, the stack is protected by Stackguard and the access results in a memory fault. Similar problems arose in the application using the *TI* RTOS. Specific functions like for example `Queue_put(Queue_Object* obj, Queue_Elem* elem)` will possibly store data onto the stack of a task not currently executing, which will result in a memory fault.

There are two ways to solve such a memory access violation. The first way to resolve this problem would be to make use of the privilege levels defined in the *ARMv7-M* architecture. The two available privileged levels are *unprivileged (user mode)* and *privileged*. (ARM 2010, p. 512 [5]) The usage of the privilege levels can help when configuring the MPU access permissions for the individual regions. If the tasks are executing unprivileged and the RTOS is executing privileged, the MPU regions can be configured to allow access from the RTOS and deny access from other tasks. In the applications under consideration, the tasks are executing privileged. It is possible to configure *Keil RTX* to execute tasks unprivileged. The *TI* RTOS kernel (*SYS/BIOS*) does not support unprivileged task, respectively user mode in general, for performance reasons. (Texas Instruments Inc. 2020, [19]) However, although possible to configure *Keil RTX* for unprivileged tasks, this is no suitable solution for this applications because of the needed adjustments that are accompanied by such a change. In the current implementations, the tasks are directly accessing system resources like for example the *Nested Vectored Interrupt Controller (NVIC)*. This access

would not be possible if the tasks would be executing unprivileged. To keep the needed adjustments at a minimum, the second way to resolve this problem has been used. In this approach, the Stackguard is explicitly disabled for the specific RTOS functions that need to access the task stacks and immediately enabled afterwards. This is a pragmatic solution which needs minimal adjustments and can be performed if the RTOS is available as source code.

5.1.5 Conclusion of the Integration

The integration of the Sheaperd library into commercial software applications was successful. The encountered challenges regarding Stackguard resulted from the missing possibility to provided user-created stacks to the task creation process. The newer CMSIS v2 API, as well as other RTOS implementations like *TI* RTOS or *FreeRTOS*, provide this functionality. Nevertheless, due to the fact that the used operating systems were available as source code, the necessary stack alignment adjustments could be implemented. (See 5.1.1) Another challenge resulted from the notification of task switches. The notify macro for task switches of the *Keil RTX* RTOS is called when the task switch is requested. At this point, Stackguard can not adjust the MPU region, as the RTOS still needs to access the stack after the macro has been executed. A custom macro that is called, after all stack accesses have been performed, has been inserted to solve this problem. (See 5.1.2) The same problem occurred with the task-switch callback of the *TI* RTOS. The stack of the switched out task was accessed after the callback. The MPU activation has been delayed until the `Task_enter` function is called. A similar problem arose when data was directly stored onto the stack of another task and the MPU was still active for the associated region. This has been solved by manually enabling and disabling the MPU. (See 5.1.4) An alternative and more secure approach is to use the unprivileged execution level for tasks and the privileged execution level for the RTOS. With this approach, the MPU configuration can be more granular and the RTOS can have access to the stacks during task switches and data delivery, but the other tasks are still isolated. Making use of the available execution levels can help to secure an application in general, and should be considered as an option when creating new applications or during the refactoring of existing ones. The Sheap allocator was added manually by replacing the existing allocations. The automation of this process can be an additional feature of the Sheaperd Eclipse plugin (see 5.2). The array assertion has been done automatically using the Sheaperd Eclipse plugin. No unmet assertions have been reported by the Sheaperd library. (See 5.1.3)

5.2 Sheaperd Design Decisions and Outlook

The Sheaperd library consists of the Sheap allocator, the memory protection module, the Stackguard, which makes use of the memory protection module and the Sheaperd Eclipse plugin. This section takes a look at the different parts of the library, provides information about specific design decisions, and

gives an outlook about further improvements.

Sheap Allocator

The goal of the Sheap allocator implementation was to provide a simple implementation which is not necessarily trimmed to minimal memory usage. The main purpose of the allocator is to assist in the debugging process. The implementation emphasizes this with the extended memory layout. (See 4.1.2) The layout comes with additional memory usage, but also provides the developer with additional information when debugging. The same argument can be made for the `alignment offset` in the memory block layout. (See figure 7) The alignment offset occupies additional memory, but helps to some out of bound writes when freeing a memory block. The extended memory layout has the benefit that it stores the program counters of the callers during allocations and deallocations. This does not necessarily mean that any allocation and deallocation can be traced. Consider a function `allocMyStruct`, which is used to allocate memory for a specific structure and is called by different other functions to obtain that structure. The extended memory layout would always record the program counter inside the `allocMyStruct` function and could not determine where the call originated. A future version of the Sheap allocator could contain an even wider extended memory layout, which also stores a specific, maybe configurable, number of bytes of the stack trace when performing allocations and deallocations. With that additional information, it is more likely to exactly determine the source of the allocation. As already mentioned in 4.1.1, additional memory allocation strategies can be implemented in future version of the Sheap allocator. The best strategy can then be selected depending on the use case at hand.

Stackguard and Memory Protection

Stackguard makes use of the memory protection module. The memory protection module is used to configure the MPU. (See 4.2.1) The MPU implementation follows the *ARMv7-M* architecture. This comes with the already mentioned limitations regarding the base address and the MPU region size. (See 4.2.1) The *ARMv8-M* architecture has, besides other new features, also refined the MPU. Chapter 5.3 discusses some of these new features and changes, as well as their implications regarding the Sheaperd library.

Sheaperd Eclipse Plugin

The Sheaperd Eclipse plugin provides commands in the context menu of *C* projects to automatically insert and remove array assertions. The integration into existing applications showed that a similar feature to automatically exchange existing allocation and deallocation calls (`malloc`, `calloc` and `free`) with the associated sheap functions (`sheap_malloc_lr`, `sheap_calloc_lr` and `sheap_free_lr`) would be a useful extension. Furthermore, the Sheaperd executor view can be used with the *Addr2Line* executable to find the associated

file location of memory addresses. The array assertion has the potential for improvement. In the current version (3.5.0), the assertion fails on shadowed array variables.

5.3 ARMv8-M Architecture Outlook

This thesis mainly focuses on Cortex-M models: M3, M4 and M7 and therefore on the *ARMv7-M* architecture. The *ARMv7-M* architecture comes with some limitation, especially regarding the MPU. (See 4.2.1) Arm recognized this problem and, among other things, adjusted the MPU implementation in the *ARMv8-M* architecture. Figure 27 shows the differences in creating a memory region for the example address range of `0x3BC00-0x80400`. While in the *ARMv7-M* architecture multiple regions are needed to cover the mentioned memory area, in the *ARMv8-M* architecture a single region can cover the complete area. In the *ARMv8-M* architecture, an MPU region can be of arbitrary size with a granularity of 32 bytes. Due to this additional flexibility, the *ARMv8-M* architecture removed subregions and does not allow MPU regions to overlap. (ARM 2016, p. 15 [6])

Another features of the *ARMv8-M* architecture that could be of interest for the Sheaperd library is the *TrustZone technology*. The TrustZone technology is an additional feature that partitions the complete memory into secure and non-secure sections. Only secure software is allowed to access the secure memory sections. The processor is in secure state when it is executing code from secure memory. Otherwise, the processor is in the non-secure state. Processors that support the TrustZone technology are accompanied by additional hardware to configure the secure and non-secure memory sections. Changing from the non-secure to the secure stat is only possible via explicitly defined entry points. The intention of this feature is to isolate specific, security critical, parts of the software from common application code. For example, a secured firmware, a licensed RTOS, device driver libraries or a certified communication stack can be isolated from the running application code. Figure 28 shows an example use case of the TrustZone technology. The execution starts with the secured firmware and will hand the control to the RTOS at some point. During execution, the non-secure software can use specific entry points to access the GUI or the protocol stack library. This can help to protect the software against intentional and unintentional corruption or manipulation. This feature does not replace the MPU but can be used additionally. (ARM 2018, p. 6 ff. [11]; Yiu 2016, p. 3 ff. [25])

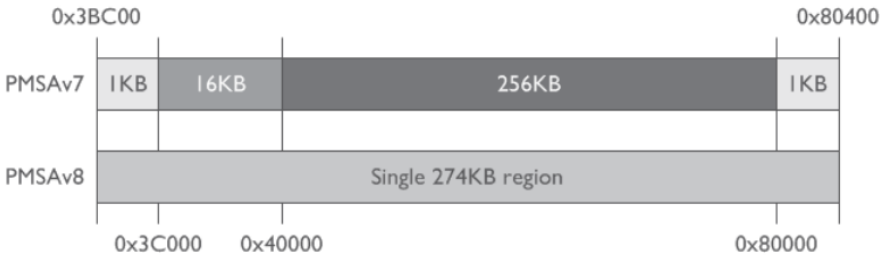


Figure 27: Comparison of the creation of memory regions in *ARMv7-M* and *ARMv8-M* architecture. A region of size 274 KB at the base address `0x3BC00` should be created. In the *ARMv7-M* architecture, multiple regions need to be allocated to cover the mentioned region. (PMSAv7) The *ARMv8-M* architecture allows MPU regions of any size at a granularity of 32 bytes and can simply create a single region. (PMSAv8) (Source: ARM 2016, p. 15 [6])

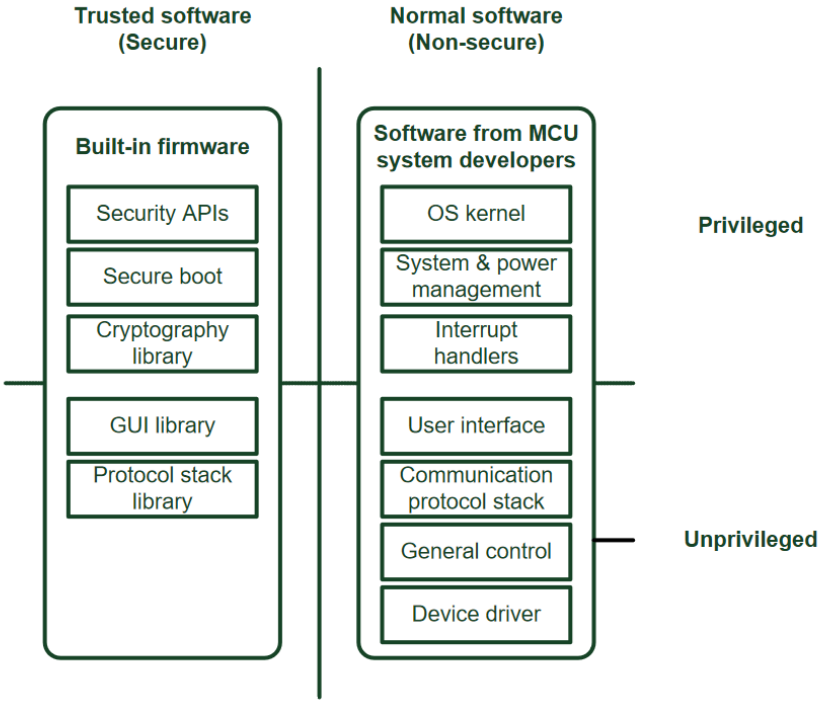


Figure 28: An example segmentation of the secure and non-secure memory sections using the TrustZone technology. The secure section contains security critical components that need to stay unaltered. The non-secure memory sections contain the common application code, which does not need to be secured. (Source: Yiu 2016, p. 3 [25])

Glossary

Abstract syntax tree is a tree used to represent the hierarchical structure of a source code file. Figure 29 shows a simple example of an AST describing a single function which obtains the length of a *C*-style string.

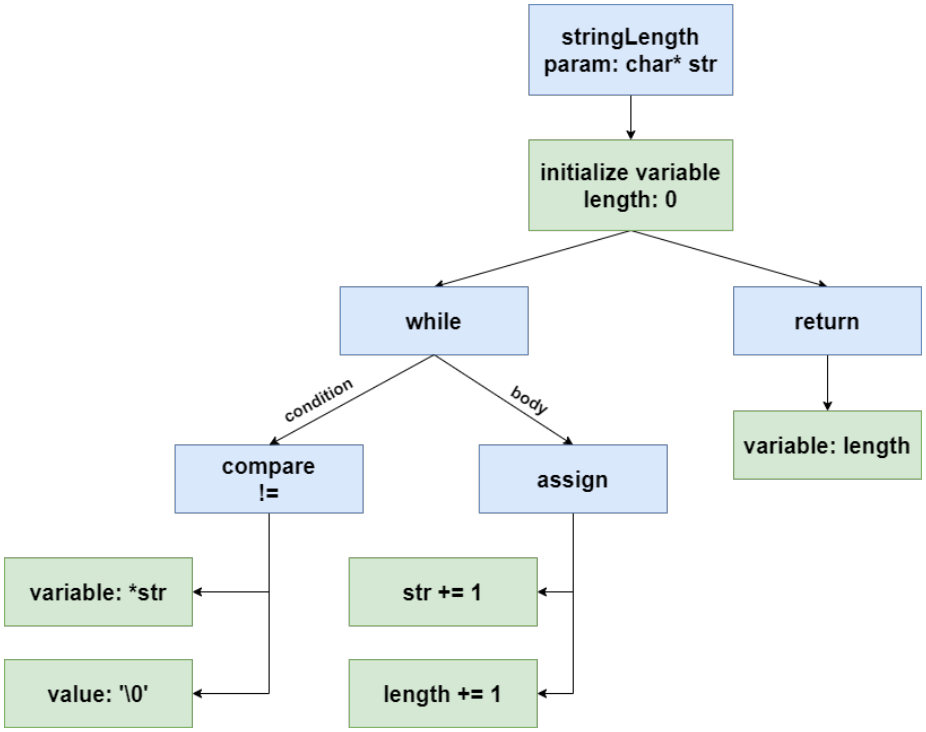


Figure 29: An example AST describing a *C* function which obtains the length of a provided string.

9, 43–45

Advanced RISC Machines (ARM) develops processor architectures and designs cores following the architectures and licenses it to other companies. 1–5, 53

Coalescing means to combine into a single group or thing. In the context of computer science it is usually used to describe the action of merging/combining of free adjacent memory blocks into one big block. Figure 30 shows an example borrowed from [18] where process memory is freed. In the scenarios b, c and d the freed memory has been coalesced with the free adjacent memory.

Figure 30: Different scenarios where a process is terminated and the associated memory is freed. The dashed area represents the free memory. (Source: Tanenbaum and Bos 2015, p. 192 [18]

i, 17, 18, 21, 22

Common Microcontroller Software Interface Standard provides interfaces for processor and peripherals, real-time operating systems, and middleware components. The goal is to simply reuse of software and reduce the time to market for new devices. 46, 51

Common Weakness Enumeration (CWE) is a community-developed list of software and hardware weakness types. The project is backed and supported from the MITRE Corporation, US-CERT and the National Cyber Security Division of the U.S. Department of Homeland Security. 1, 25, 43

Constant folding describes a compiler technique. During compile time, the compiler recognises that the value of an expression is constant and replaces the expression with the identified constant. (Aho et al. 2014, p. 536/591 [1]) i, 11

Copy propagation describes a compiler technique. After a copy statement $a = b$, b should be used wherever possible instead of a. One advantage of the copy propagation is that after it has been performed, the copy statement may be dead code. (Aho et al. 2014, p. 591 [1])
Figure 31 provides an example.

```
int main(int argc, char* argv[]) {  -->     int main(int argc, char* argv[]) {
    int i = argc;                   -->         int i = argc;
    int j = 10 + i;                 -->         int j = 10 + argc;
    /* i is a copy of argc */       -->         /* copy propagation has been performed */
}                                   -->     }
```

Figure 31: Copy propagation: the copy statement `'int i = argc;'` turns into dead code after the copy propagation.

i, 11

Dangling pointer is a pointer which does not point to a valid address in memory. A pointer that is used after it has been free is a kind of dangling pointer. A pointer to a variable which ran out of scope is another example for a dangling pointer. See figures 32 and 33 for examples.

```
int main(int argc, char* argv[]){
    int* pValue = malloc(sizeof(int) * 10);
    if(pValue == NULL){
        // Perform out of memory handling
    }
    free(pValue);
    *pValue = 11;        // Dangling pointer: pValue has already been freed
}
```

Figure 32: Dangling pointer: A pointer being used after it has been freed

```
int main(int argc, char* argv[]){
    int* pValue = NULL;
    {
        int value = 10;
        pValue = &value;
    }
    *pValue = 11;        // Dangling pointer: value has already run out of scope
}
```

Figure 33: Dangling pointer: A pointer being used after the variable it points to ran out of scope

1

Dead code elimination describes a compiler technique. During compile time, the compiler recognises that parts of the code are not reachable and omits them when generating code. (Aho et al. 2014, p. 591 [1])
Figure 34 provides an example.

```
int main(int argc, char* argv[]) {
    int i = 10;
    bool check = false;
    if (check) {         // will never resolve to true
        i++;             // will never be executed
    }
}
```

Figure 34: Dead code example: the body of the if condition will never be executed. Therefore the compiler can omit it when generating code.

i, 11

Direct memory access is a feature that enables hardware to directly access memory. Due to the direct access the CPU is not needed for this operation and can perform other tasks. If no DMA is used, the CPU would be occupied during the memory access operation. 38

Dynamic binary analysis (DBA) is the process of analyzing a possibly instrumented binary at runtime. The main advantage of this technique is that no source code is required. (Nethercote 2004, p. 2f [15]) 6

Dynamic binary instrumentation (DBI) is the process of additional statements to the binary representation of a software dynamically with the intent to gather information during the execution. (Huang 2009, p. 163 [9]) 6–8

Integrated development environment (IDE) is a software application intended for software development. It usually contains a source editor, build and debugging tools. It is often extensible using plugins. 4, 5, 43

Intermediate representation (IR) is a lossless representation of a source language. It is mostly used by compilers and virtual machines and supports optimisation as well as the modularisation of the compilation process into front end and back end. (Aho et al. 2014, p. 4f [1])(See figure 35)

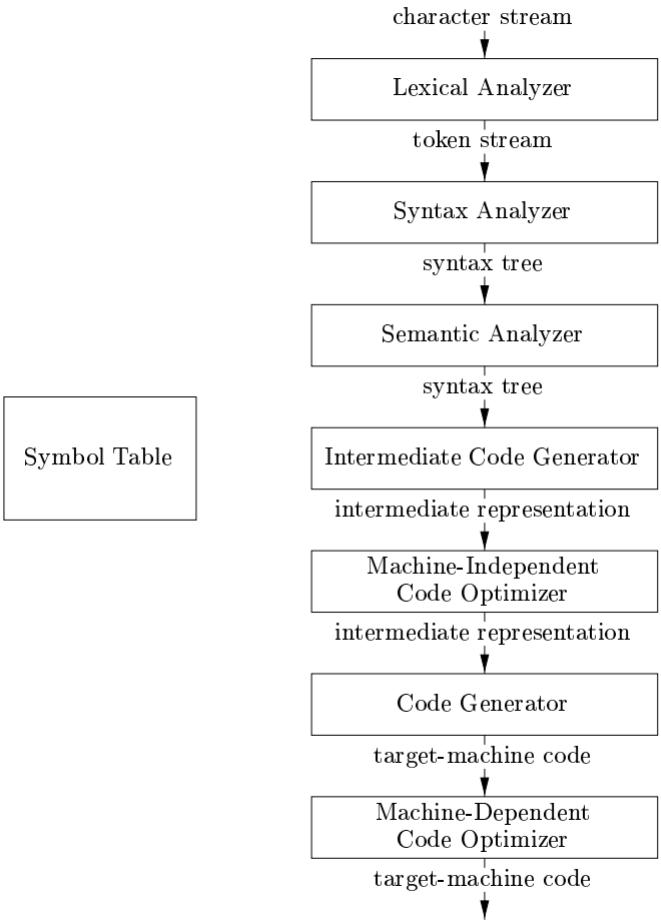


Figure 35: Phases of compilation with intermediate representation.
(Source: Aho et al. 2014, p. 4f [1])

Line of code (LOC) is a metric used to compare software by counting the number of lines of source code needed to create it. 1

Microcontroller unit describes an integrated circuit that consists, alongside one or multiple CPUs, of memory and programmable I/O peripherals. Figure 36 shows the differences to a CPU. 38, 49

Real-time operating system (RTOS) is used for the development of real time applications. Real time is often misinterpreted in this context. It

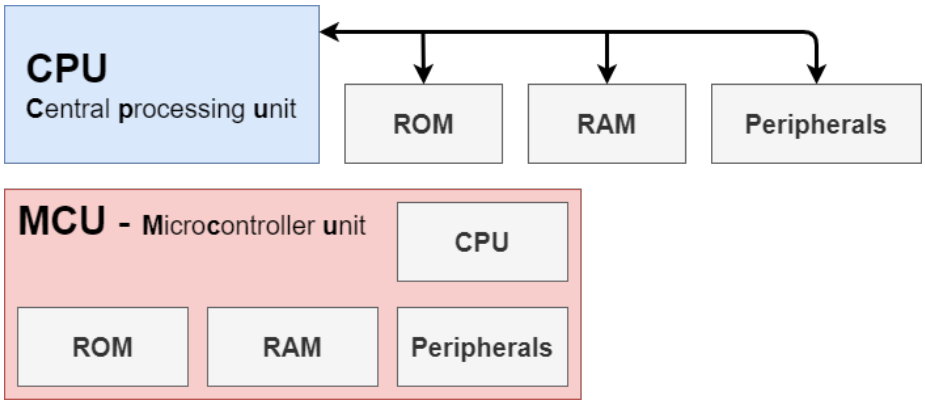


Figure 36: Distinction between a CPU and a MCU.

simply means that a task is completed in a predetermined time frame. 3, 4, 12, 14, 33, 40, 41, 46, 47, 49–51, 53

TIOBE Index is an monthly updated index which ranks the most used programming languages. Multiple search engines and other sources are utilized to keep the index updated. See https://www.tiobe.com/tiobe-index/ for more information. (accessed 23 March 2021) i, 1, 2

References

[1] Alfred V. Aho et al. *Compilers: principles, techniques, and tools*. en. 2. ed, Pearson new intern. ed. OCLC: 858019306. Harlow: Pearson, 2014. ISBN: 978-1-292-02434-9.

[2] Pansy Arafa et al. "Debugging Behaviour of Embedded-Software Developers: An Exploratory Study". en. In: *arXiv:1704.03397 [cs]* (Apr. 2017). arXiv: 1704.03397, p. 5. URL: http://arxiv.org/abs/1704.03397 (visited on 07/06/2021).

[3] ARM Limited. "ARM Compiler armcc User Guide". en. In: (2014), p. 1008. URL: https://documentation-service.arm.com/static/5eb946b50f1c1e0dae6ee21e?token= (visited on 06/15/2021).

[4] ARM Limited. "ARM Compiler toolchain Assembler Reference". en. In: (2013), p. 643. URL: https://documentation-service.arm.com/static/5ea68b849931941038ded96e?token= (visited on 06/18/2021).

[5] ARM Limited. "Armv7-M Architecture Reference Manual". en. In: (2010), p. 858. URL: https://documentation-service.arm.com/static/606dc36485368c4c2b1bf62f?token= (visited on 06/15/2021).

[6] ARM Limited. "Armv8-M Memory Protection Unit (MPU)". en. In: (2016), p. 39. URL: https://documentation-service.arm.com/static/5ef61f08dbdee951c1ccdd48?token= (visited on 07/04/2021).

[7] Pavan Kumar Chittimalli and Vipul Shah. "GEMS: A Generic Model Based Source Code Instrumentation Framework". en. In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. Montreal, QC: IEEE, Apr. 2012, pp. 909–914. ISBN: 978-1-4577-1906-6 978-0-7695-4670-4. DOI: 10.1109/ICST.2012.195. URL: https://ieeexplore.ieee.org/document/6200109/ (visited on 07/07/2021).

[8] Jon Erickson. *Hacking: the art of exploitation*. 2nd ed. OCLC: ocn175218338. San Francisco, CA: No Starch Press, 2008. ISBN: 978-1-59327-144-2.

[9] J. C. Huang. *Software error detection through testing and analysis*. OCLC: ocn263498238. Hoboken, N.J: John Wiley & Sons, 2009. ISBN: 978-0-470-40444-7.

[10] Sang Moo Huh and Woo-Je Kim. "The Derivation of Defect Priorities and Core Defects through Impact Relationship Analysis between Embedded Software Defects". en. In: *Applied Sciences* 10.19 (Oct. 2020), p. 18. ISSN: 2076-3417. DOI: 10.3390/app10196946. URL: https://www.mdpi.com/2076-3417/10/19/6946 (visited on 05/01/2021).

[11] ARM Limited. "Arm TrustZone Technology for the Armv8-M Architecture". en. In: (2018), p. 26. URL: https://documentation-service.arm.com/static/5f873034f86e16515cdb6d3e?token= (visited on 07/04/2021).

[12] Steve McConnell. *Code complete*. en. 2nd ed. Redmond, Wash: Microsoft Press, 2004. ISBN: 978-0-7356-1967-8.

[13] MITRE. *CWE - 2020 CWE Top 25 Most Dangerous Software Weaknesses*. URL: https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html (visited on 05/30/2021).

[14] Jan Mußler. "A Generic Binary Instrumenter and Heuristics to Select Relevant Instrumentation Points". In: (2010), p. 91. URL: https://core.ac.uk/download/pdf/34893191.pdf (visited on 04/13/2021).

[15] Nicholas Nethercote. "Dynamic Binary Analysis and Instrumentation or Building Tools is Easy". PhD thesis. Trinity College, University of Cambridge, 2004.

[16] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". en. In: *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)* (2007), p. 12. URL: https://valgrind.org/docs/valgrind2007.pdf (visited on 04/28/2021).

[17] STMicroelectronics. "STM32 Cortex-M4 MCUs and MPUs programming manual". en. In: (2020), p. 262. URL: https://www.st.com/resource/en/programming_manual/dm00046982-stm32-cortex-m4-mcus-and-mpus-programming-manual-stmicroelectronics.pdf (visited on 06/19/2021).

[18] Andrew S. Tanenbaum and Herbert Bos. *Modern operating systems*. Fourth edition. Boston: Pearson, 2015. ISBN: 978-0-13-359162-0.

[19] Texas Instruments Inc. *TI-RTOS Kernel (SYS/BIOS) User's Guide: Device Addendum*. 2020. URL: https://software-dl.ti.com/simplelink/esd/simplelink_cc13x2_26x2_sdk/5.20.00.52/exports/docs/tirtos/sysbios/docs/Device_Addendum.html#cortexm-operating-modes-and-stack-usage (visited on 08/21/2021).

[20] Valgrind Developers. *Valgrind - Memcheck: a memory error detector*. URL: https://valgrind.org/docs/manual/mc-manual.html (visited on 04/13/2021).

[21] Valgrind Developers. *Valgrind - The Valgrind Quick Start Guide*. URL: https://valgrind.org/docs/manual/quick-start.html#quick-start.mcrun (visited on 04/13/2021).

[22] Valgrind Developers. *Valgrind - Using and understanding the Valgrind core*. URL: https://www.valgrind.org/docs/manual/manual-core.html (visited on 04/28/2021).

[23] Paul R. Wilson et al. “Dynamic storage allocation: A survey and critical review”. en. In: *Memory Management*. Ed. by Gerhard Goos et al. Vol. 986. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 1–116. ISBN: 978-3-540-60368-9 978-3-540-45511-0. DOI: 10.1007/3-540-60368-9_19. URL: http://link.springer.com/10.1007/3-540-60368-9_19 (visited on 05/25/2021).

[24] Joseph Yiu. *The definitive guide to ARM® Cortex®-M3 and Cortex-M4 processors*. Third edition. OCLC: ocn859555920. Amsterdam: Elsevier, Newnes, 2014. ISBN: 978-0-12-408082-9.

[25] Joseph Yiu. “The Next Steps in the Evoluation of Embedded Processors for the Smart Connected Era”. en. In: (2016), p. 10. (Visited on 07/04/2021).