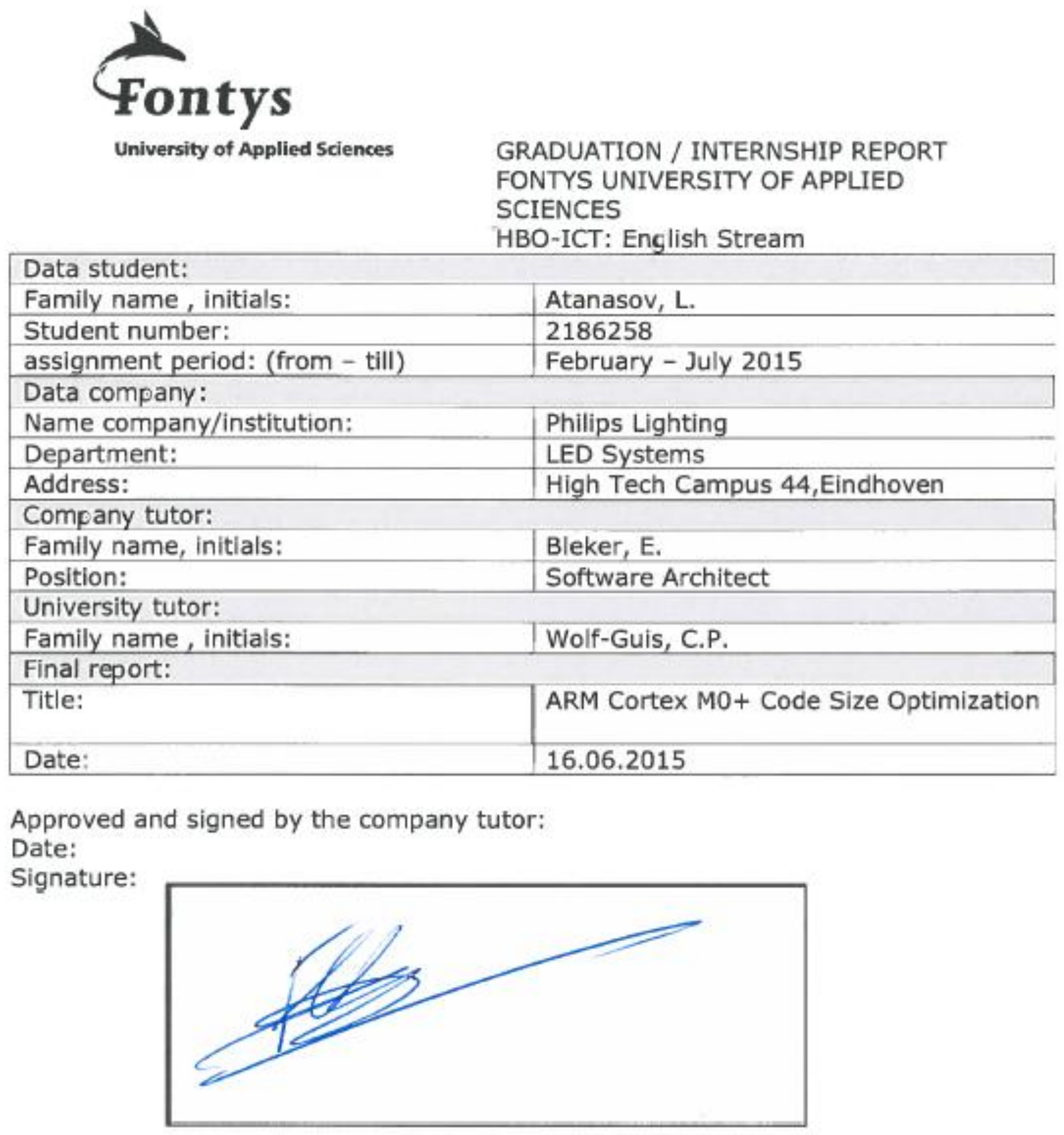
# ARM CORTEX M0+ CODE SIZE OPTIMIZATION

FY 2015

*Philips Lighting graduation project*

*Because (code) size matters*

Author: Lyubomir Atanasov



Fontys
University of Applied Sciences

GRADUATION / INTERNSHIP REPORT
FONTYS UNIVERSITY OF APPLIED SCIENCES
HBO-ICT: English Stream

| | |
|---|---|
| Data student: | |
| Family name , initials: | Atanasov, L. |
| Student number: | 2186258 |
| assignment period: (from – till) | February – July 2015 |
| Data company: | |
| Name company/institution: | Philips Lighting |
| Department: | LED Systems |
| Address: | High Tech Campus 44,Eindhoven |
| Company tutor: | |
| Family name, initials: | Bleker, E. |
| Position: | Software Architect |
| University tutor: | |
| Family name , initials: | Wolf-Guis, C.P. |
| Final report: | |
| Title: | ARM Cortex M0+ Code Size Optimization |
| Date: | 16.06.2015 |

Approved and signed by the company tutor:
Date:
Signature: [illegible]

## Preface

This report documents the work done during the graduation project at Philips Lighting which took place during my final fourth year of education, pursuing a Bachelor's degree in ICT&Software at Fontys University of Applied Science. The goal of the report is to give a clear overview of the tasks completed during that period and the approach used for the completion of the research assignment.

## Acknowledgments

I would like to express my sincere gratitude to my colleagues, for the challenging five months spent at Philips Lighting. I would like to acknowledge all those who were assisting me when needed.
I would also like to thank Erik Bleker (Company Mentor/Software Architect) guiding me through the assignment and Rudi Block (Project Leader) for helping me keep track of the project progress.
Lastly, I would like to thank Mrs. Wolf-Guis (University Mentor), who showed a great interest in my project and spent admirable efforts in attempt to assist me in improving the quality of this final report.

### Abbreviations

| | |
|---|---|
| *RSA*: | Reusable Software Architecture |
| *DMA*: | Direct Memory Access |
| *LED*: | Light-emitting diode |
| *IDE*: | Integrated development environment definition |
| *RAM*: | Random Access Memory |
| *ROM*: | Read Only Memory |

### Terms

*Interrupt*: An interrupt is a signal, which causes the CPU to stop what it is doing and execute a separate piece of code designed by the programmer.

*EWARM*: Compiler toolchain found in IAR Workbench IDE.

*GNU*: Open-source compiler toolchain

# Contents

## *Summary*

*The graduation project was executed at Philips Lighting B.V. where among the various lighting solutions, LED drivers are being developed. Microcontrollers which are the brain of a LED driver, only have a limited memory capacity. The software capabilities drivers must provide, grow exponentially thus the memory resources will soon become insufficient. To tackle this issue, the possibility for optimizing the code for the ARM Cortex M0+ microcontroller was investigated. Two research topics were covered- increasing the code density of the program and delegating somepi software capabilities to the DMAC and Atmel EVSYS hardware components. Recommendations about applicable code reduction optimization techniques were given and their impact on the program was analyzed. Investigation on DMA and Atmel EVSYS was carried, which proved that combining both components is possible and can contribute to reducing the CPU load of the software.*

# 1 Introduction

*Because (code) size matters…*

Philips Lighting is specialized in offering professional lighting solutions for the business and consumer market. LED drivers are one of the products the company manufactures. Philips Lighting produces different types of LED drivers, based on the customer needs and requirements (indoor, outdoor etc.)

LED drivers are hardware devices which can convert incoming AC power to the proper DC voltage, and control the current flowing through the LED. Without the proper driver the LED may become unstable causing decreased performance or failure.

Microcontrollers- microcomputers incorporating the processor, RAM, ROM and I/O ports into a single package, are often employed in LED drivers. Microcontrollers can be considered as the "brain" of a LED driver. They can be programmed to provide software capabilities such as dimming, fading, color-changing and others.

There is a vast variety of different microcontroller vendors available on the market, for example: ARM, Atmel, Intel and many others. Microcontrollers, often have different characteristics such as speed, input/output pins, supply voltage and **memory size**.

Memory size is one of the important criteria in embedded software development. Microcontrollers usually provide 8/16/32 kb of memory space. Contrary to modern desktop application development, when creating a software for an embedded system, the programmer must also consider the software capabilities to be implemented versus the available memory space. In the real world increasing memory capacity adds direct costs to the final product.

This issue brings the topic of **Code Density** (size). Code Density refers to the number of assembly instructions needed to perform a requested action and the amount of space each instruction takes. The less space an instruction takes and the more work per instruction it performs, the higher achieved density of the code is.

Throughout this graduation report, different means and techniques to achieve high Code Density will be discussed. The target hardware platform will be the **ARM Cortex M0+** microcontroller. The software under optimization is the Reusable Software Architecture (**RSA**) repository, being developed at Philips Lighting.

# 2 The Company

*The world needs more light!*

Royal Philips of the Netherlands is a diversified technology company, focused on improving people's lives through meaningful innovations in the areas of Healthcare, Consumer Lifestyle and **Lighting**.

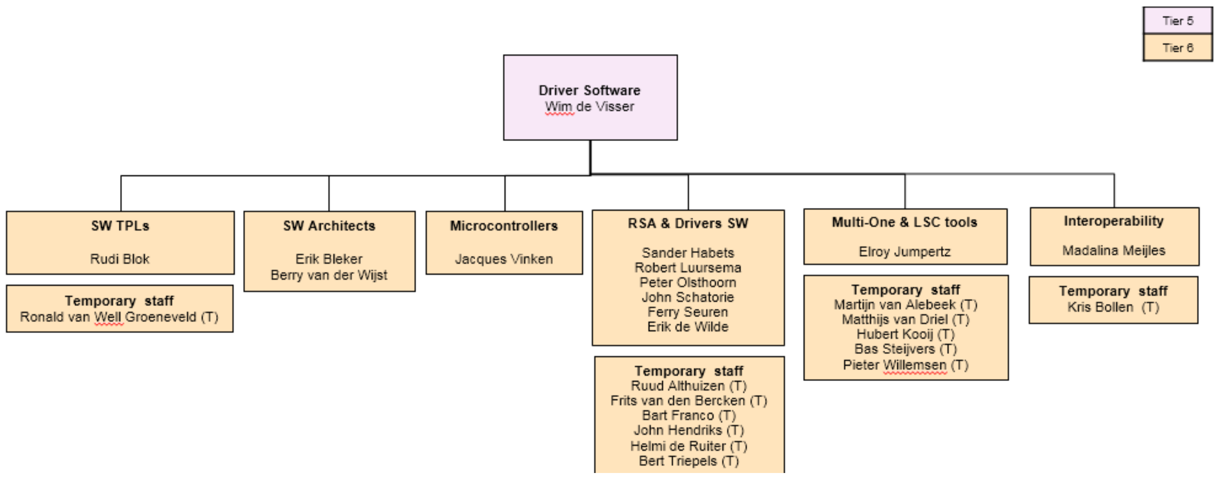
Philips Lighting, where the graduation assignment is performed, is a global market leader with recognized expertise in the development, manufacturing and application of innovative lighting solutions. Philips Lighting provides lighting solutions that include indoor and outdoor luminaries, electronics, and lighting controls. The core product sectors of the company are lighting for the home, professional lighting and LED modules.

Being the world's leader in lighting, the company is persistently promoting the switch to energy efficient solutions. With lighting compromising a significant portion of current energy consumption, the latest LED and lighting controls solution from Philips can drastically improve the overall energy performance.

Philips Lighting has manufacturing facilities in 25 countries in all regions of the world and sales organizations in more than 60 countries. The company currently employs approximately 53 000 people worldwide.

The graduation assignment was performed in the Philips Lighting- Eindhoven facility, where among the various work areas LED drivers are being developed. The RSA & Drivers SW team is responsible for the development and maintenance of the embedded software in drivers. For the purpose of the assignment the graduation intern is part of this team.

Graph 1 depicts the organizational hierarchy in Philips Lighting– Eindhoven. The specific functions and responsibilities of the RSA & Drivers SW team are discussed in the next chapters.



Graph 1

# 3 Project Overview

The following chapters discuss the project topic, the reasoning behind initiating the project and the possible benefits of its successful completion.

## 3.1 PROJECT DESCRIPTION

The Driver Software department at Philips Lighting is responsible for developing embedded software in LED electronic drivers. There are various types of LED drivers which could be categorized based on their purpose (indoor, outdoor) and based on their functional requirements (programmable, dimmable, current flow etc.)

Despite their differences LED drivers would often share similar or even the same software capabilities. To avoid unnecessary extra development effort, common functionality is thus separated into software packages. The Reusable Software Architecture (**RSA**) is responsible for developing and maintaining these packages into a single SVN Repository. The three main strong points of the RSA can be easily highlighted:

- ❖ Unified behavior between the different LED drivers.
- ❖ Increased quality coming from reusing existing well tested software.
- ❖ Decreased development cost and time as result of avoiding duplicate work effort.

With the continuously growing number of applications of LED lighting, the required software capabilities to be implemented within a single LED driver also increase. The drivers however have limited capacity, thus having software with high Code Density can sometimes make the difference in achieving the competitive edge.

The main part of the graduation assignment is to investigate the different means and techniques of achieving high Code Density of the RSA repository on a specific hardware platform – the SAM D 21 microcontroller.

## 3.2 INITIAL SITUATION

The RSA repository consists of a large number of software packages, which define common for different LED drivers, functionalities. The repository is already being used from various projects across multiple sites at Philips Lighting, targeting different types of drivers.

Up until recently, only 8 microcontrollers were being used. With the evolution of the semi-conductor industry the requirements a LED driver must fulfill increased, which led to introducing the 32-bit microcontroller– Atmel Sam D 21.

The memory capacity of the Atmel SAM D 21 is however soon going to reach its maximum and possible code optimization actions should be investigated.

## 3.3 PROJECT OBJECTIVE

The project aim is to bring insight into code optimization covering two important research fields:

- Investigate the possibility for code size reduction of the RSA repository on the ARM Cortex M0+ microcontroller. (**major part of assignment**)
- Investigate the possibility to reduce CPU load by delegating software functionality to the DMAC and Atmel EVSYS hardware components. (**secondary part of assignment**)

The research on code size should at least cover the possible optimization techniques and their applicability to the RSA. In addition the significance of the compiler in this aspect should be determined.

The research on DMA and Atmel EVSYS should bring knowledge about the two hardware components, give proposal for an applicable situation, and provide a demo application as a proof of concept.

## 3.4 PROJECT JUSTIFICATION

*"We should forget about small efficiencies, say about 97% of the time; premature optimization is the root of all evil" Donald E. Knuth, Structured Programming with go to Statements*

Achieving higher code density on the Atmel SAM D 21 microcontroller will lead to smaller binaries allowing implementing additional software capabilities without having to upgrade the hardware. For example it has been calculated, that in the current case upgrading the memory capacity will add extra 10 cents to the bill of materials. For a product like LED drivers, which is manufactured in a large scale, this is of great significance.

Delegating software functionalities to the DMA and Atmel EVSYS hardware components, will potentially reduce the CPU usage. In addition the distributed functionality, can be executed parallel to the currently performed by the CPU tasks which might lead to increased performance.

## 3.5 TOOLS & APPLICATIONS

Various software tools have been used to support the successful execution of the graduation assignment.

Throughout the assignment several development IDEs have been used (the reasoning behind using such a variety of IDEs is discussed further in the report):

- IAR Workbench
- Atmel Studio
- Keil MDK

The RSA repository requires the usage of a subversion software- which in this case is TortoiseSVN. The purpose of using sub-versioning for the RSA is discussed later in the report.

To complete the assignment good understanding of the C language and Assembly language is required – the transition between the high level C language and low level Assembly language is key to achieving high code density.

## 3.6 PROJECT RESTRICTIONS

The target hardware platform is the Atmel SAM D 21 microcontroller. The evaluation kit containing this microcontroller is the Atmel Xplained Pro, which is used for easy developing and testing.

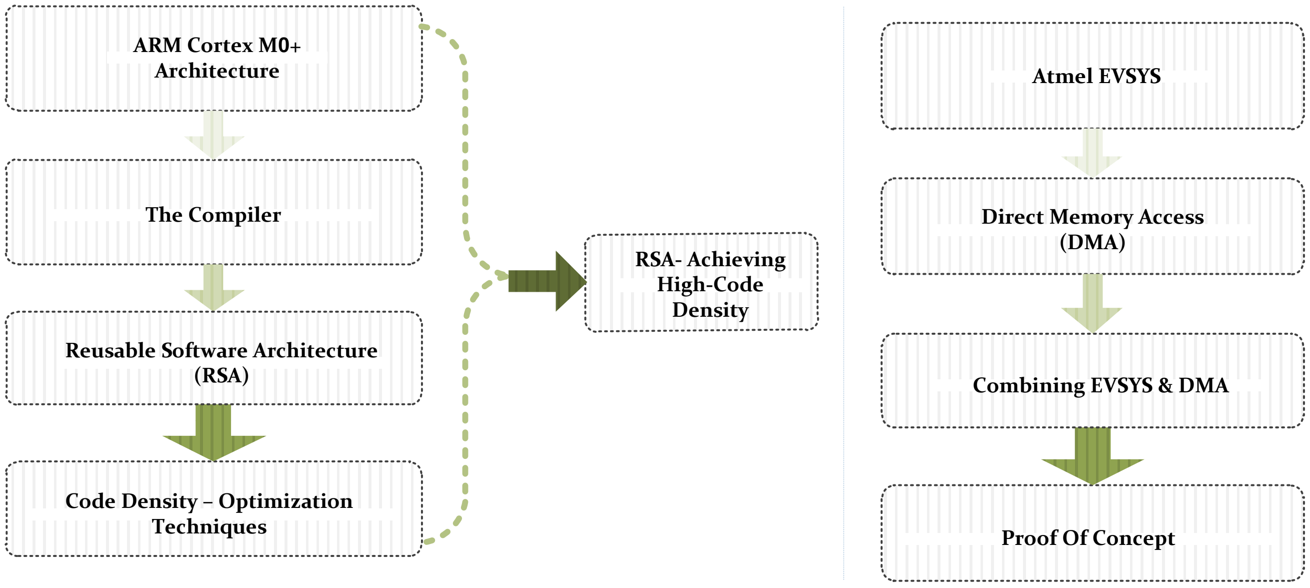
The RSA repository structure must be kept intact. If any optimizations are applied, they must not affect the software behavior. The way the RSA repository is being referenced from other projects must not change.

The readability, maintainability and portability of the code must not be affected by the applied optimization techniques- the RSA packages must be kept platform independent.

## 3.7 PROJECT EXECUTION METHODOLOGY

The project is executed in an agile SCRUM methodology. At the beginning of each sprint a planning meeting is held between the graduation intern, the Project manager and the Software architect (also company tutor) to determine the tasks for the upcoming period. Each task is assigned with human effort measured in days and a weekly plan is created. On daily bases SCRUM meetings are being held, where the previous day progress and the upcoming work are discussed.

The scope of the project covers the major research topics, which are important for code density. Eventually the applicability of the optimizations to the RSA is determined. The last phase of the project carries the research on Atmel EVSYS and DMA. The project can logically, be separated into the following research activities:


ARM Cortex M0+ Architecture
The Compiler
Reusable Software Architecture (RSA)
Code Density – Optimization Techniques
RSA- Achieving High-Code Density
Atmel EVSYS
Direct Memory Access (DMA)
Combining EVSYS & DMA
Proof Of Concept

The first part of the assignment is covered in chapter 4 and the second part is covered in chapter 5.

Graph 2

## ARM Cortex M0+ Architecture

In the beginning a study on the ARM Cortex M0+ architecture was carried. The used materials were mainly from literature (1; 2; 7; 9; 10).

## The Compiler

Research on the compiler design was carried. Plenty of materials were found on the internet. In addition information about the available ARM Cortex M0+ compilers was gathered from the official documentation of the toolchains. The RSA was then ported for the compilers, and a benchmark the achieved code density on the different compilers was done.

## Reusable Software Architecture

Knowledge about the RSA was built through reading the repository wiki - package description, architecture design and tutorials. In order to get familiar with the software functionalities and detect possible issues, the code was examined. The code size of all the RSA modules was measured and stored into a simple Microsoft access database.

## Code Density- Optimization Techniques

Research on the available code optimization techniques was carried- both in the common case and such specific for the ARM Cortex M0+ microcontroller. The discovered techniques were verified, and those which were found inapplicable to the ARM Cortex M0+ were discarded.

## RSA- High Code Density

Based on the output of the previous activities recommendations for achieving high code density of the RSA were given in the following directions: compiler optimization settings and applicability of the discovered optimization settings.

## Atmel EVSYS

Study on the Atmel Event System was carried. The used materials were mainly from the SAM D 21 datasheet (4) and Atmel SAM D21 Xplained user guide (5). Some demo examples were tried for further exploration.

## Direct Memory Access

Study on Direct Memory Access was carried. The used materials were mainly from the SAM D 21 datasheet and reference manual. Some demo examples were tried for further exploration.

## Combining EVSYS & DMA

Beneficial application of combining DMA & EVSYS was discovered and designed. A working prototype was developed as a proof of concept.

# 4 Code Size Optimization

## 4.1 CODE DENSITY INTRODUCTION

**Code Density** “the amount of space that an executable program takes up in memory” is an important criteria for microcontrollers which have limited amount of memory. To illustrate the principle of code density the following example can be taken, where the commonly used memcpy method compiled without any optimizations applied, targeting an 8bit microcontroller:

memcpy C code

```
void memcpy(void *dest, void
*source, int count_bytes)
{
    char *s, *d;

    s = source; d = dest;
    while(count_bytes--)
       {
        *d++ = *s++;
       }
}
```

memcpy Assembly code

```
mov r0, count_bytes
mov r1, s
mov r2, d
/***Enter loop***/
loop:
ldrb r3, [r1]
strb [r2], r3
mov r3, 1
add r1, r3
add r2, r3
sub r0, r3
cmp r0, 0
bne loop
/***Exit loop***/
```

We can count that in this case the memcpy method takes **11** assembly instructions. This can amount of memory can be considered as **low** density inefficient code. The same code compiled on a modern 32bit microcontroller with compiler optimizations turned on may look something like this:

memcpy C code

```
void memcpy(void *dest, void
*source, int count_bytes)
{
    char *s, *d;

    s = source; d = dest;
    while(count_bytes--)
       {
        *d++ = *s++;
       }
}
```

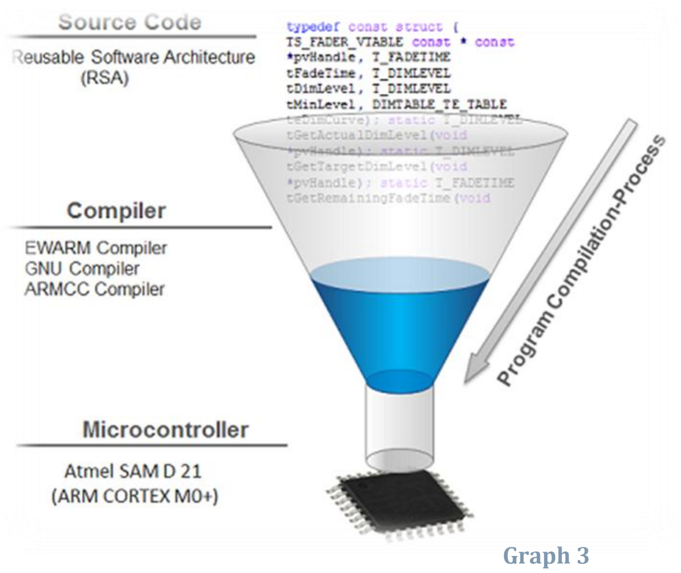
memcpy Assembly code

```
movl r0, count_bytes
movl r1, s
movl r2, d
/***Enter loop***/
loop: ldrb r3, [r1++]
strb [r2++], r3
subs r0, r0, 1
bne loop
/***Exit loop***/
```

This time the compiled memcpy method only takes **7** instructions. This source code can be considered to have higher **code** density with a clear advantage of **4** instructions compared to the previous example.

The compiler produces different output, based on the **source code** and **microcontroller architecture**. This is a result of the different transformations occurring during the program compilation process- various **compilers** may also generate altered output code. As a consequence of these observations, the factors which directly affect code density can be broadly divided into the following categories:

- ❖ The Source Code– in the current case the Reusable Software Architecture (RSA) is to be optimized.
- ❖ The Compiler- there are several compiler toolchains which can be used to build the RSA on the ARM Cortex M0+ microcontroller: EWARM, GNU and ARMCC.
- ❖ The (device) Target- platform on which the code is to be executed is the Atmel SAM D21 microcontroller.



Graph 3

**The microcontroller** is the hardware device which is responsible for executing the program. Microcontrollers are built according to different architectures and often possess different characteristics such as:

- ❖ Registers purpose, number and size.
- ❖ Instruction set.
- ❖ Data type, size and addressing.

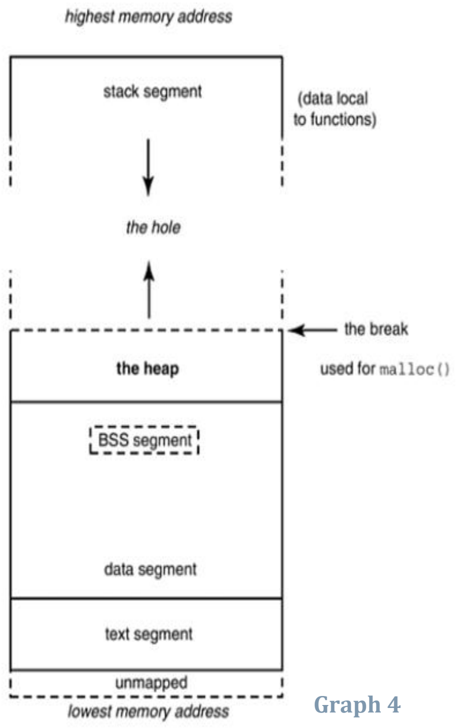
These microcontroller features have a direct impact on the program size and performance. An overview of the SAM D 21 (ARM Cortex M0+) microcontroller is given in chapter 4.2.

**The compiler** is the software which is responsible for the translation of high-level source code into binary. Compilers offer different optimization settings which can significantly decrease the memory footprint of the program. The compiler design, compilation process and Cortex M0+ compilers are discussed in chapter 4.3.

**The source code** can be written in different high-level programming languages. Those programming languages often provide concepts for code structure maintenance and reusability (e.g. OO programing, functional programming) which also lead to variations in the output assembly code. The program might have also been optimized for a specific target. A good overview of the RSA is given in chapter 4.4.

When a C program is run, the executable image is loaded into the memory of the microcontroller in an organized manner, called memory layout. This memory layout is organized into different segments.

Based on the **code** expression and **microcontroller architecture**, the **compiler** decides which memory segment should be occupied. In the common case the memory layout is defined by the following pattern:



Graph 4

- ### Text Section

  The text section contains the (assembly) instructions of the program. Constant variables are also stored in this section.

- ### Data Section

  The data section consists of global and static data which is initialized when declared.

- ### BSS Section

  The bss section, the same as data section, consists of global and static variables. The data is however not initialized when declared.

- ### Stack Section

  The stack section is the default storage location for variables which are local to a function. It stores various pointers which are required for the execution of the program. Stack is created at the beginning of the method, and emptied when it is completed.

- ### Registers

  Registers are memory locations which are part of the CPU. As a result access to registers is fast and very efficient. Local variables which are frequently used, are often stored in registers.

When the compiler allocates code to memory, it has to consider the characteristics of the target hardware in order to produce an efficient executable image. For example the number and purpose of registers, varies across the various microcontrollers, thus in some cases the stack will be used more heavily.

It is also important that the code gives clear and correct hints to the compiler- based on the type and size of a variable, the compiler may allocate the variable in a different memory segment or registers.

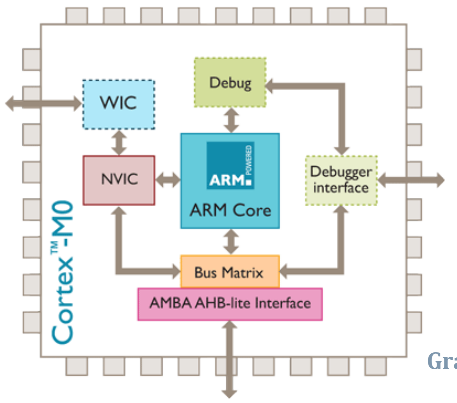
To summarize the compiler allocates “code” in memory, based on the microcontroller architecture specifications and the source code.

## 4.2 ARM CORTEX M0+ OVERVIEW

The ARM Cortex M0+ microcontroller is designed to handle C code, in a highly efficient manner. When developing software, the programmer would usually not need any specialized knowledge about this MCU. However, to completely understand the optimization techniques discussed further in the report, one should also be familiar with fundamentals about the ARM Cortex M0+ architecture. This chapters gives a brief overview of the microcontroller architecture, register set and memory model. For complete information about the processor please refer to the Cortex M0+ Technical Reference Manual (2).

### 4.2.1 ARM Cortex M0+ Architecture

The Cortex M0+ is a 32bit RISC processor. The processor understands Thumb and Thumb-2 code and provides support for either little-endian or big-endian data access. It has an AMBA AHB-Lite interface and include an NVIC component which supports up-to 32 external interrupt inputs with configurable priority.

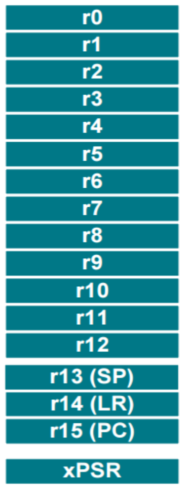


Graph 5

### 4.2.2 Registers Summary

The ARM Cortex M0+ has 13-general purpose registers. The microcontroller only have 32bit wide registers. The registers are defined as follows:

- ❖ R0-R7 (general purpose registers).
- ❖ R8-R12 (general purpose registers).
- ❖ R13 used to hold the dual stack pointers for efficient implementation of RTOS.
- ❖ R14 link register used for returning from function calls.
- ❖ R15 Program Counter (PC) register is used hold the address of the next instruction to be executed.
- ❖ Program status register (PSR) contains instruction results such as zero and carry flags interrupt mask register.



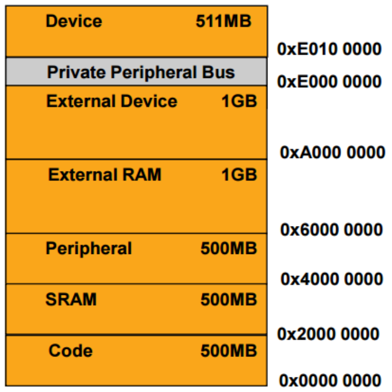
Graph 6

### 4.2.3 Memory Model

The Cortex M0+ has a default memory space that provides up-to 4 GB of addressable memory. The division of the memory sections is presented on Graph 7.

The microcontroller only supports aligned data- all types must begin at a specific address. The data-aligned requirements for the different types are presented below:

- Type `char*` must be aligned on address divisible by 4
- Type `long` must be aligned on address divisible by 4
- Type `int` must be aligned on address divisible by 4
- Types `short` must be aligned on address divisibly be 2
- Type `char` and `_Bool` must be aligned on address divisible by 1.
- Type `float` must be aligned on address divisible by 4.
- Type `double` must be aligned on address divisible by 8.
- Type `long long` must be aligned on address divisible by 8.

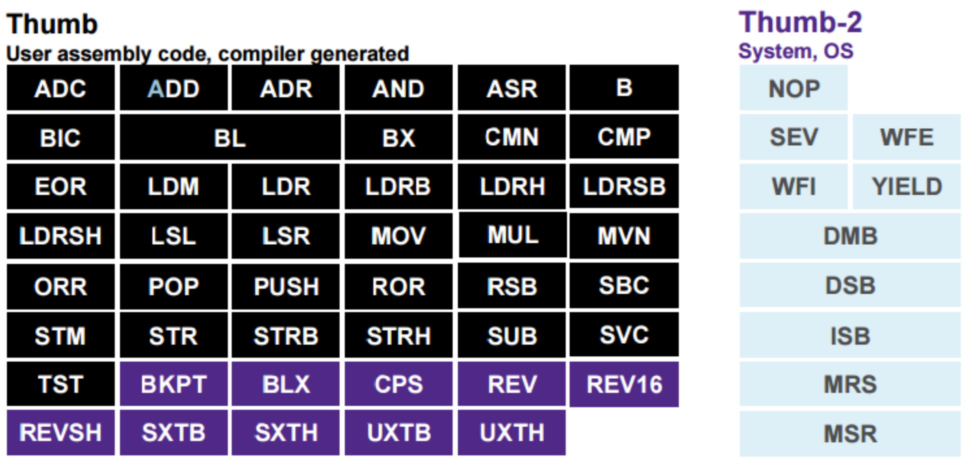


Graph 7

Any attempt to access unaligned data will cause a hard fault exception. The purpose behind aligned data is that the microcontroller can efficiently access all data types. For example the microcontroller can access all word (32bit) aligned data types with a single assembly instruction. The topic of efficient memory alignment is continued further in the next chapters of the document.

### 4.2.4 Instruction Set

The microcontroller supports the ARMv6-Thumb instruction set and the Thumb-2 extension. All Thumb instructions are 16-bit long and all Thumb-2 instructions are 32bit. The Thumb data processing instructions operate on full 32-bit values and use full 32-bit addresses for data access and instruction fetches.


Thumb
User assembly code, compiler generated
ADC ADD ADR AND ASR B
BIC BL BX CMN CMP
EOR LDM LDR LDRB LDRH LDRSB
LDRSH LSL LSR MOV MUL MVN
ORR POP PUSH ROR RSB SBC
STM STR STRB STRH SUB SVC
TST BKPT BLX CPS REV REV16
REVSH SXTB SXTH UXTB UXTH
Thumb-2
System, OS
NOP
SEV WFE
WFI YIELD
DMB
DSB
ISB
MRS
MSR
Graph 8

## 4.3 THE COMPILER OVERVIEW

Modern embedded systems are often based on a microcontroller. The microcontroller is merely a piece of mechanical device without any built-in logic, so it must be programmed and controlled by software. This hardware device can only interpreter instructions in the form of electric charge or what is refered to as binary logic. Instead of spending long hours to write code consisting of only 0s and 1s the **compiler** is used to translate (compile) high-level language source code into binary code.

To understand why and how the compiler affects code density, one should be familiar with the transformations through which the source code goes, during the compilation process. In this chapter some fundamental for the compiler topics are discussed:

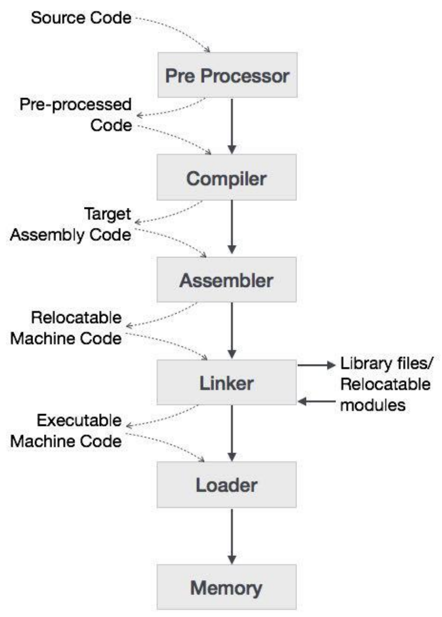
- ❖ Compiler Toolchain- the different tools involved in the build process
- ❖ Compilation Process- from C code to binary.
- ❖ Code Optimization- improvements to the code, performed during the compilation process.

### 4.3.1 Compiler Toolchain

Compiler toolchains (build-tools) are a set of programming tools used for developing software applications. Toolchains also provide software libraries with commonly required functionality. In general a software toolchain consists of the following tools:

#### ❖ PREPROCESSOR

The preprocessor transforms the source code which will later serve as an input to the compiler. The preprocessor is responsible for handling files inclusion, macro-processing and conditional compilation.



Graph 9

#### ❖ COMPILER

The compiler transforms high-level programming language source code to a low-level language (e.g. assembly language or machine code). The compiler performs different types of operations for example: lexical analysis, syntax analysis, semantic analysis, parsing, **code generation** and **code optimization**.

#### ❖ ASSEMBLER

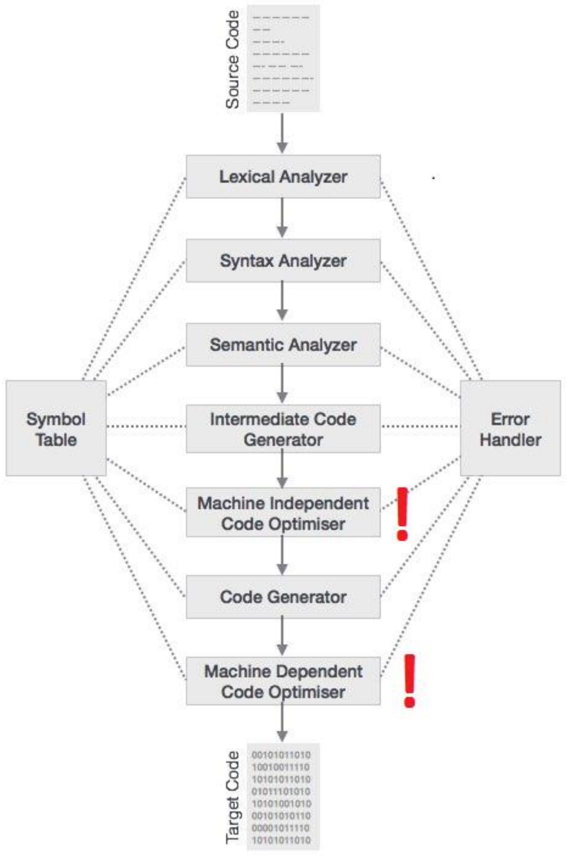
The assembler is a type of compiler which transforms assembly language source code to machine level code. Assembly language is human readable but it can typically be mapped one-to-one with the machine language code.

#### ❖ LINKER

The linker is a computer software which links and merges one or more objects files into a single executable program. Another important function of the linker is assigning addressed to the sections used by the applications, based on a specified linker configuration file. Further, the linker eliminates duplicate and unused code sections, before generating the final application image.

### 4.3.2 Compilation Process

The compilation process will usually follow a predefined sequence of phases. Each stage uses input from the previous stage, applies transformations, and feeds its output to the next stage. Compilation process will often execute in the following sequence:



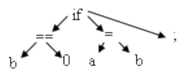
Graph 10

**The Lexical Analyzer** reads the code source character by character and converts it into a sequence of lexical tokens. To illustrate this process take a look at the following example:

| C Code | Token after lexical analysis |
|---|---|
| `int a = 123;` | `int (keyword), a (identifier), = (operator), 123 (constant) and ; (symbol)` |

**The Syntax Analyzer** checks whether the output produced by the previous phase satisfies the rules implied by the context free grammar (CFG) and generates a parse tree. The following example is used to illustrate this process:

| if | ( | b | == | 0 | ) | a | = | b | ; |
|---|---|---|---|---|---|---|---|---|---|



**The Semantic Analyzer** checks output parse tree from the previous phase for semantic errors (type checking etc.) and collects the type information for code generation. The semantic analyzer outputs an annotated syntax tree.

**The Intermediate Code Generator** generates an intermediate code representation of the source code. This code is generally architecture independent, but it is close to the level of machine code. Intermediate code provides abstraction between the high-level language code and machine code.

**The Machine-Independent Code Optimizer** uses the intermediate code representation to apply code optimizations which do not require any knowledge about the target platform. The optimizing compiler is discussed in the next chapter.

**The Code Generator** maps the optimized intermediate code representation to the target machine language. The output code can be executed by the target machine. In addition to the basic conversion the code generators also performs:

- **Instruction selection-** selects the appropriate instructions.
- **Instruction scheduling-** arranges the instructions in the optimal order.
- **Register allocation-** allocates variables to the processor registers.

**The Machine-Dependent Code Optimizer** uses generated machine code as an input and applies target-specific optimizations. The optimizing compiler is discussed in the next chapter.

### 4.3.3 The Optimizing Compiler

As part of the compilation process the optimizing compiler tries to apply various transformation techniques in order to improve the efficiency of the code, by making it consume less resources. The compiler would usually perform optimizations in respect to either achieving smaller (high code density) code or better performance (CPU utilization etc.) It is common that some optimization techniques will involve a size versus speed tradeoff.

The most important rule the optimizing compiler must follow during the optimization process is that the optimized code cannot at any case change the semantics of the program. As a result the compiler must often approach the possible optimizations conservatively – it cannot make assumptions whether the result of the program will change or not.

Most modern compilers also have an interface through which the programmer can configure the optimization settings. Compilers designed for embedded software would usually provide optimization in two directions- optimize for **speed** or optimize for **size**. As mentioned earlier there is often a speed-to-size tradeoff.

Compilers also often provide different levels of optimization .With each level the compiler will optimize more aggressively. Selecting the highest optimization level will significantly increase the compilation time and produce code which is relatively hard to debug.

In addition to selecting an optimization level, the programmer can also enable or disable specific optimizations. Compilers usually provided quick on/off switches for the following optimizations:

- Live-dead analysis and optimization
- Dead code elimination
- Redundant label elimination
- Redundant branch elimination
- Code hoisting
- Peephole optimization
- Some register content analysis and optimization
- Common subexpression elimination
- Code motion
- Static clustering
- Cross jumping
- Loop unrolling
- Function inlining

The previous chapter explained that optimizations mainly occur in two phases during the compilation process- before and after code generation. Optimizations can be categorized into two types: machine-independent and machine-dependent optimizations.

##### *4.3.3.1 Machine-Independent Optimizations*

Machine-independent optimizations are those which the compiler can perform regardless the target processor/microcontroller. Such techniques will usually try to improve the flow of the code. More information about machine-independent optimizations accompanied by examples can be found in Appendix *B*.

##### *4.3.3.2 Machine-Dependent Optimizations*

Machine-dependent optimizations are those applied with internal knowledge of the target machine architecture. The compiler will take advantage of microcontroller specific information for the instruction set and memory layout (registers, stack, heap etc.) to produce more efficient code. More information about machine-dependent optimizations accompanied by examples can be found in Appendix *A*.

#### 4.3.4 Standard C Library

Most compilers also provide a library which contains common programming functionalities (mathematical computations, memory allocation, string handling etc.). Such libraries, often have a significant memory footprint, thus compiler for embedded software will sometimes offer an optimized version.

### 4.3.5 ARM Cortex M0+ Compilers

In the previous chapter the various transformations through which the code goes during the compilation process were examined. It is than normal that the different compilers will produce different output binaries. Often a specially designed for a specific microcontroller architecture compiler, will produce more efficient code with higher density. In order to discover whether a particular compiler performs better in achieving denser code, benchmarking the RSA on the available in market compilers, is needed.

There are several compiler toolchains which can be used to compile source code for the ARM Cortex M0+ microcontroller:

- **IAR-EWARM** commercial compiler toolchain found in IAR Work-Bench (6).
- **GNU Tools for ARM Embedded Processors** open-source compiler toolchain, can be integrated in various IDEs (11).
- **ARM Compilation Tools** commercial compiler toolchain found in Keil MDK (3).

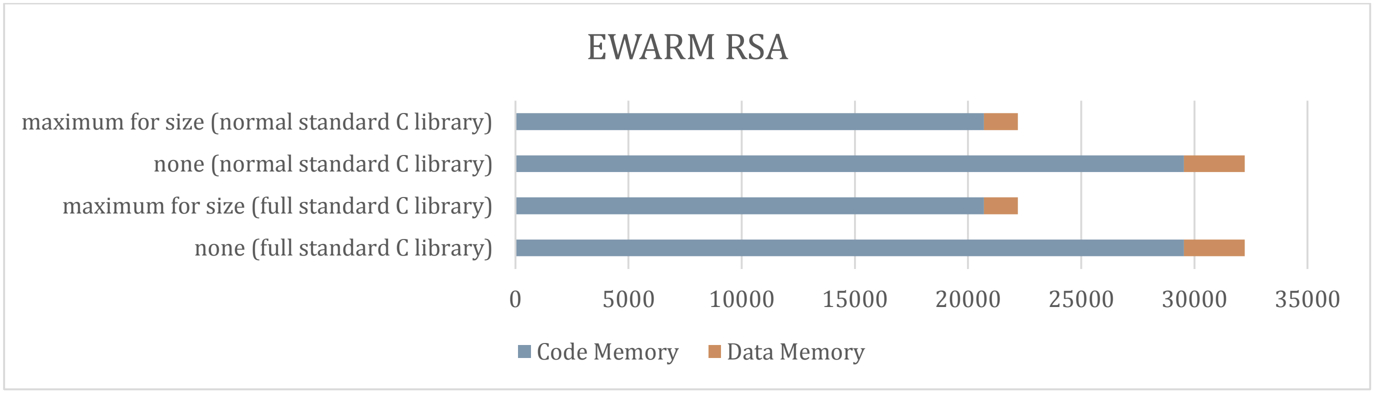
The following chapters examined the program size of the RSA execute on ARM Cortex M0+ microcontroller, compiled with different build-tool chains.

#### *4.3.5.1 IAR-EWARM RSA Code Size*

The results of building the AtSamD21Driver project with EWARM compiler are presented below. From the table one can see, that there is a significant code reduction of **8860** bytes when optimizing for size and total reduction of **10032**. There is also no difference between compiling with the full standard C library or the normal standard C library.

| Optimization Level | Code Memory | Data Memory | Total |
|---|---|---|---|
| none (full standard C library) | 29534 | 2698 | 32232 |
| maximum for size (full standard C library) | 20698 | 1502 | 22200 |
| none (normal standard C library) | 29534 | 2698 | 32232 |
| maximum for size (normal standard C library) | 20698 | 1502 | 22200 |

Table 1



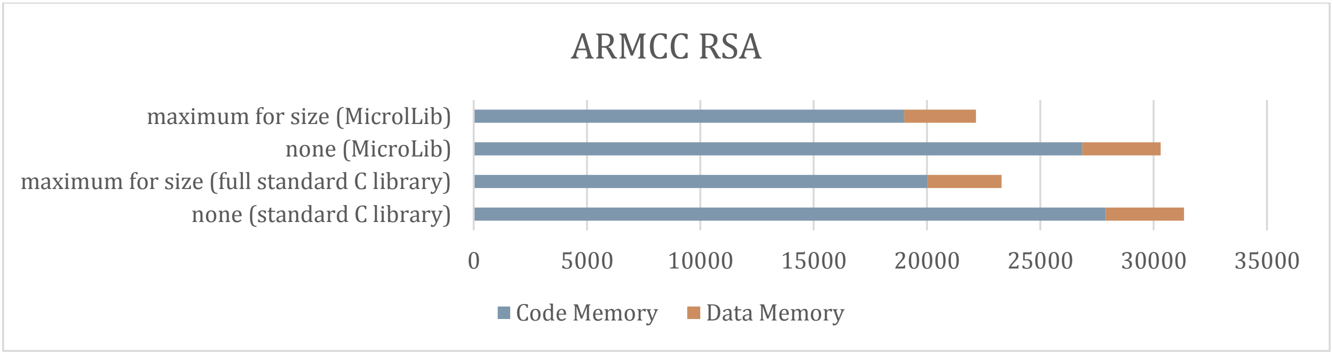
Graph 11

#### *4.3.5.2 ARM Compilation Tools RSA Code Size*

The results of building the AtSamD21Driver project with ARMCC compiler are presented below. From the table one can see, that there is a significant code reduction of **8896** bytes when optimizing for size and total reduction of **9176** bytes. Compiling with the microlib library bring some notable reduction of **1128** bytes (code+ data memory).

| Optimization Level | Code Memory | Data Memory | Total |
|---|---|---|---|
| none (standard C library) | 27884 | 3452 | 31336 |
| maximum for size (full standard C library) | 20020 | 3268 | 23288 |
| none (MicroLib) | 26852 | 3452 | 30304 |
| maximum for size (MicroLib) | 18988 | 3172 | 22160 |

**Table 2**



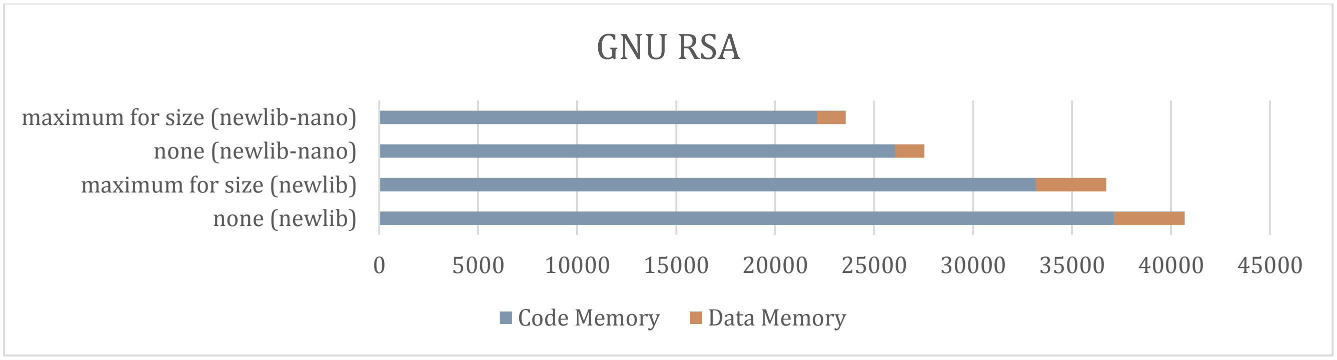
**Graph 12**

#### *4.3.5.3 GNU Tools RSA Code Size*

The results of building the AtSamD21Driver project with GNU compiler are presented below. From the table we can see the significant code and total memory reduction when optimizing for size. Using newlib-nano seems to be crucial for achieving good code density while showing difference of **11056** bytes.

| Optimization Level | Code Memory | Data Memory | Total |
|---|---|---|---|
| none (newlib) | 37119 | 3572 | 40691 |
| maximum for size (newlib) | 33164 | 3560 | 36724 |
| none (newlib-nano) | 26064 | 1472 | 27536 |
| maximum for size (newlib-nano) | 22108 | 1460 | 23568 |

**Table 3**



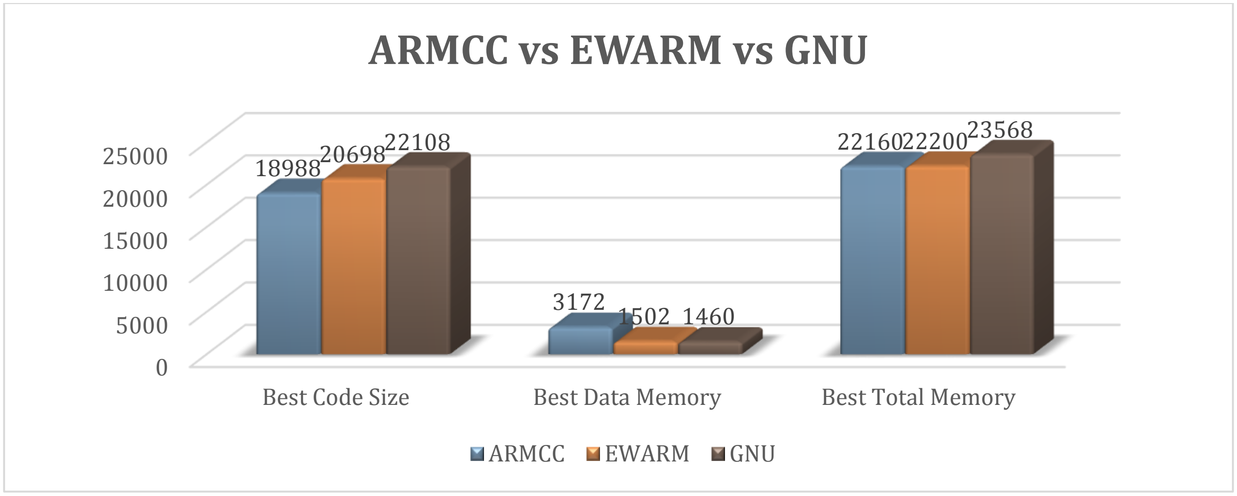
Graph 13

### 4.3.5.4 ARMCC vs EWARM vs GNU

The best code size, data memory and total memory the toolchains achieved are shown below. From the data we can make the conclusion that they seem to give quite similar results. The commercial toolchains ARMCC and EWARM give almost identical results when considering Total Memory. Their optimizations settings are easy to configure and do not require in-depth knowledge of the toolchain. GNU toolchain is slightly behind but, considering that it is a free toolchain the results are satisfying

| Compiler | Best Code Size | Best Data Memory | Best Total Memory |
|---|---:|---:|---:|
| ARMCC | 18988 | 3172 | 22160 |
| EWARM | 20698 | 1502 | 22200 |
| GNU | 22108 | 1460 | 23568 |

Table 4



Graph 14

## 4.4 RSA OVERVIEW

### 4.4.1 RSA Introduction

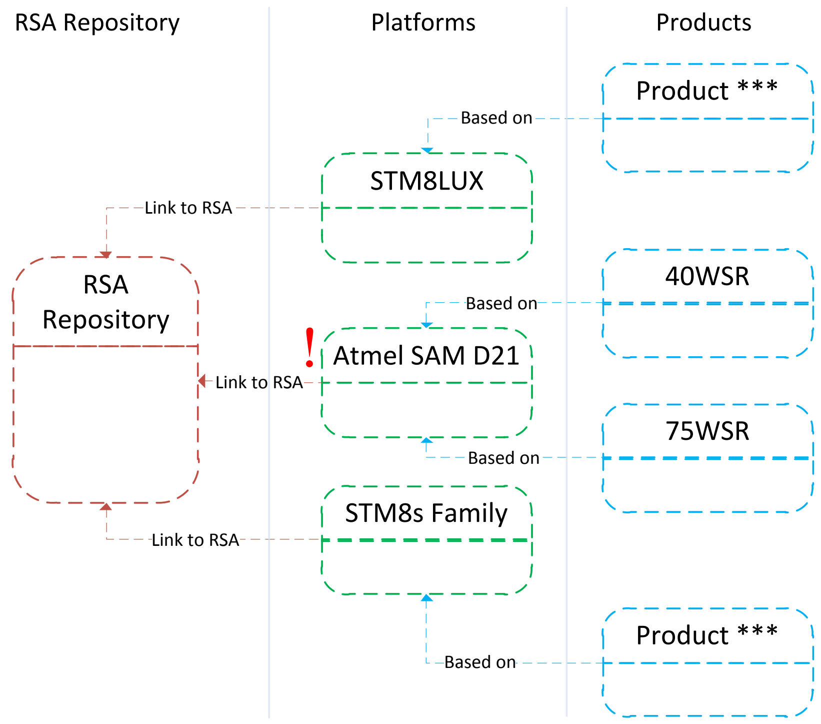
The RSA is a software architecture which serves as a foundation for multiple projects. It administers the reuse of software packages for various products within Philips Lighting.

The RSA aims to be hardware independent by providing abstraction from the different types of microcontrollers used within LED drivers. The RSA also provides support for multiple MCUs for example the: Atmel SAM D 21, ATTiny 44/84 and STM8LUX microcontrollers. Such support is defined into a **platform** which can be a single instance for multiple **products**. To clarify the process the following example is taken:

"*Driver for a 40W LED bulb, based on the Atmel SAM D 21 microcontroller must be developed*".

We have however previously developed a driver for a 75W LED bulb using the same microcontroller, for which purpose we have already defined a platform for the Atmel SAM D 21. As a result we can reuse this platform for our new 40W driver.

The platform is based on software packages coming from the RSA repository. If in the future we want to use a different microcontroller for the 40W/75W LED drivers, there is no need to do any changes on the software packages coming from the RSA- only the microcontroller-specific functionality needs to be changed. The concept about reusing software packages from the RSA is depicted on Graph 15.



Graph 15

### 4.4.2 RSA Architecture Design

The RSA architecture is organized according to the “Layered Application” design pattern. Each layer has packages which consist of multiple modules. Projects which want to reuse functionality from the RSA must only reference to packages and not modules.

The four defined layers are the application layer, the service layer, the hardware layer and the utility layer. Graph 16 depicts the communication between the layers.

#### ❖ APPLICATION LAYER

The Application Layer is the top layer of the RSA. It provides autonomous applications (collection of services). Different projects often have to provide their own implementation for this layer.

#### ❖ SERVICE LAYER

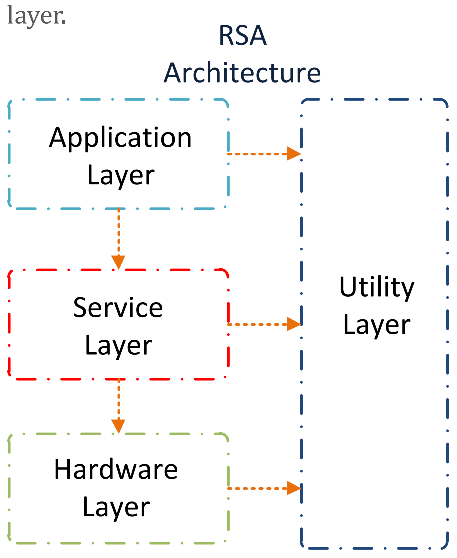
The Service Layers provides functionality which can be used by applications for example, fading service, communication through the DALI interface and others.

#### ❖ HARDWARE LAYER

The Hardware Layer adds abstraction between the functionality in the Service Layer and the actual hardware on which the program is to be executed. All modules from the Hardware Layer must be able to operate without knowledge about the hardware.

#### ❖ UTILITY LAYER

The Utility Layer provides functionality which is common for all layers. This way duplicate code across the other layers is avoided.



Graph 16

### 4.4.3 RSA Implementation Details

The RSA repository is programmed in C language. The architecture has been in development for several years already, so it has been initially design for an 8bit microcontroller, thus it is also optimal for such. The 8bit microcontrollers have of course a natural word length 8 bits. As a result they operate most efficiently with 8bit data types.

Currently, the RSA only uses small data types (8/16 bits), in order to reduce memory footprint. For example variables of type `char` are used as loop counters in all cases. Local variables, methods and parameter types are also mostly `char` and `unsigned short int` types.

## 4.5 EFFICIENT C FOR ARM CORTEX M0+

In chapter 4.3.3 the various optimizations, applied by the compiler were discussed. It was also noted that the compiler, must sometimes take a conservative approach and not apply an optimization, in order to maintain the program behavior. This leaves certain open-ends to optimize, for which the programmer is responsible.

In addition chapter 4.4.3, explained the RSA was initially implemented and optimized for an 8bit microcontroller. In combination with the knowledge about the ARM Cortex M0+ architecture (chapter 4.2) and the compiler week points, it might be possible to perform further optimizations.

Such techniques are often considered as micro optimizations, and should only be applied when the software is already efficient on:

- Architectural Level
- Algorithm Level
- Design Level

For example using a bad sorting algorithm or having a poorly designed application with duplicate functionalities, will almost always have a bigger impact on code density, than microcontroller-specific optimizations.

The following chapters discuss techniques for achieving high code density on the ARM Cortex M0+ microcontroller. While techniques are presented in a fairly precise light, they all follow the same logical path- exploit specific for the ARM Cortex M0+ features. To achieve higher code density the main targets are:

- Reducing the stack usage.
- Better register allocation.
- Efficient memory storage.
- Efficient memory access.

In order to fully understand the concept of the discussed techniques some knowledge about C and Assembly language is advised. Familiarity with the microcontroller characteristics discussed in chapter 4.2 is also required. If not mentioned explicitly examples are compiled with the EWARM Compiler on optimization level- **High** (**optimize for code size**).

### 4.5.1 Data Structures Memory Layout

#### *4.5.1.1 The Padding Issue*

In 4.2.3 it was shown that the compiler aligns variables on specific address location in order to access data more efficiently. This often means that padding (unused empty bytes) has to be added between the variables. The following examples shows a possible layout of variables in memory.

| Expected Layout | Actual Layout |
|---|---|
| `int *a; /*takes 4 bytes*/` | `int *a;      /*takes 4 bytes*/` |
| `char b  /*takes 1 byte*/` | `char b       /*takes 1 byte*/` |
| `int c;  /*takes 4 byte*/` | `char pad[3] /* 3 bytes padding*/` |
| | `int c;       /*takes 4 bytes*/` |

To fulfil the alignment requirements the compiler adds 3 bytes of padding in order to make sure `int c` will be aligned on an address divisible by four.

#### *4.5.1.2 Padding in Structures*

One could now also expect that the compiler will align the members of a structure. This is true, but members of the structure, however have the alignment of their widest scalar member (the variable which requires largest alignment). Let's consider the following example:

| Expected Layout | Actual Layout |
|---|---|
| `typedef struct foo1 {` | `typedef struct foo1 {` |
| `    short int m1;/*takes 2 bytes*/` | `    short int m1;/*takes 2 bytes*/` |
| `    int m2;      /*takes 4 bytes*/` | `    char pad[2]  /* 2 bytes padding*/` |
| `    short int m3;/*takes 2 bytes*/` | `    int m2;      /*takes 4 bytes*/` |
| `}foo1Struct;` | `    char pad[2]  /* 2 bytes padding*/` |
| | `    short int m3;/*takes 2 bytes*/` |
| | `}foo1Struct;` |

The member of *foo1Struct* which requires highest alignment is `int m2`. As a result all other members of the structure will have to start on an address divisible of 4. The compiler will thus add padding to fulfill those requirements. The size of the structure would than become not the expected 8 bytes but 12 bytes.

#### *4.5.1.3 Reducing Padding in Structures*

It is possible to reduce the extra padding inside a structure by manually rearranging its members. This process is often referred to as Structure Packing. Applying this techniques is easy and involves reordering the structure members in a decreasing alignment sequence- members with the highest alignment requirements are declared on top and those with lower alignment are declared after them. Apply this simple technique to foo1Struct produces the following result:

Expected Layout

```
typedef struct foo1 {
    int m2;       /*takes 4 bytes*/
    short int m1;/*takes 2 bytes*/
    short int m3;/*takes 2 bytes*/

}foo1Struct;
```

Actual Layout

```
typedef struct foo1 {
    int m2;       /*takes 4 bytes*/
    short int m1;/*takes 2 bytes*/
    short int m3;/*takes 2 bytes*/

}foo1Struct;
```

By re-arranging the members of the structure the alignment requirements are now fulfilled. The compiler does not have to add extra padding. As a result the foo1Struct size is reduced from **12** bytes to **8** bytes.

### 4.5.2 Unaligned Data Access

In chapter 4.2.3 it was shown that the ARM Cortex-M0+ microcontroller does not support unaligned data access. It would however sometimes happen that the programmer would have to use data which is unaligned or with different natural alignment than the Cortex-M0+. This could for example happen during communication between the microcontroller and external interface.

When accessing unaligned data the compiler has to use additional assembly instruction, to impose proper alignment. This leads to significant increase in the code size of the program. To illustrate the issue the following example is taken.

Aligned structure access C code

```
struct foo1 {
      int num1;
      double num3;
      short int num3   ;
}Structure;
main()
{
struct foo1 alignedStrucutre={1,2,3};
int a= alignedStrucutre.num1;
int b= alignedStrucutre.num2;
int c = alignedStrucutre.num3;
volatile int sum = a+b+c;
}
```

Aligned structure access assembly code

```
SUB       SP,SP,#+4
MOVS      R0,#+7
STR       R0,[SP, #+0]
MOVS      R0,#+0
ADD       SP,SP,#+4
BX        LR
```

In the first example all structure members are naturally aligned by the compiler. Now the assembly output consists of only **6** instructions.

Unaligned structure access C code

```
#pragma pack(push,1)
struct foo1 {
      int num1;
      double num3;
      short int num2;

}Structure;
#pragma pack(pop)
main()
{
struct foo1 alignedStrucutre={1,2,3};
int a= alignedStrucutre.num1;
int b= alignedStrucutre.num2;
int c = alignedStrucutre.num3;
volatile int sum = a+b+c;
}
```

Unaligned structure access assembly code

```
PUSH     {LR}
SUB      SP,SP,#+20
ADD      R0,SP,#+4
Nop
ADR.N    R1,`?<Constant {1, (2.0), 3}>`
MOVS     R2,#+16
BL       __aeabi_memcpy4
LDR      R0,[SP, #+4]
MOVS     R1,#+12
ADD      R2,SP,#+4
LDRSH    R1,[R2, R1]
LSLS     R1,R1,#+1
ADDS     R0,R0,R1
STR      R0,[SP, #+0]
MOVS     R0,#+0
ADD      SP,SP,#+20
POP      {PC}
```

For the second example the `#pragma pack(push,1)` compiler directive is used to force the compiler remove the padding required for the data types alignment. As a result access to members of the structure will be unaligned. The compiler thus generates extra assembly instructions (byte shifting operations and others) to ensure that the alignment requirement is fulfilled. In this case there is a total of **17** instructions which is **11** more than accessing a properly memory aligned structure.

### 4.5.3 Function Register Allocation

#### *4.5.3.1 Parameter Passing*

In chapter 4.2.2 it was shown the ARM Cortex-M0+ microcontroller has a numbers of registers which serve different purpose.
The microcontroller follows the (Procedure Call Standard for the ARM Architecture, 2012) which would impose that registers **R0-R3** are available for passing arguments to a function, and register **R0** is used for passing the result of a function.

In case more than **4 (**word-sized**)** parameters are passed, the last one will be spilled (placed) in stack memory. The concept is shown in the next example, where **5** parameters are passed to a function. In the example one can observe that the compiler places the last **4** arguments in the registers, when the first one n1 is stored in stack memory.

The problem with keeping parameters in stack is that extra instructions for storing and retrieving the variable are required. For the register allocation to be efficient, the programmer should aim to limit the parameters to **4**. If parameters more are required, he can group them in one structure and pass a pointer to it.

| Function with 5 parameters C code | Function with 5 parameters assembly code |
|---|---|
| `int foo( int n1, int n2,` | `PUSH    {R7,LR}` |
| `         int n3, int n4,int n5)` | `MOVS    R0,#+5` |
| `{` | `STR     R0,[SP, #+0]//spill n1 in stack` |
| `  return n1+n2+n3+n4+n5;` | `MOVS    R3,#+4` |
| `}` | `MOVS    R2,#+3` |
| `void main()` | `MOVS    R1,#+2` |
| `{` | `MOVS    R0,#+1` |
| ` int res=foo(1,2,3,4,5);` | `BL      foo` |
| `}` | `POP     {R0,PC}` |

### 4.5.3.2 *Local Variables*

The ARM Cortex-M0+ microcontroller can hold up-to **8** local variables in its registers. Analogically to the previous chapter if more local variables are declared they will be stored in stack memory. This concept is shown in the next example, where **9** local variables are being declared. In the example one can see that the compiler places the last **8** variables in the microcontroller registers when the first **n1** is stored in stack memory.

Storing and reading variables in stack memory requires extra assembly instructions. In order to maintain efficient register allocation, the programmer should aim to declare **8** variables at most. In case more are needed he can try to split the function.

| Function with 9 local variables C code | Function with 9 local variables assembly code |
|---|---|
| `void foo()` | `PUSH    {R3-R7}` |
| `{` | `MOVS    R0,#+1` |
| `int` | `STR     R0,[SP, #+0]//spill n1 in stack` |
| `n1=1, /**Stored in stack**/` | `MOVS    R0,#+2` |
| `n2=2,n3=3,n4=4,` | `MOVS    R1,#+3` |
| `n5=5,n6=6,n7=7,` | `MOVS    R2,#+4` |
| `n8=8,n9=9;` | `MOVS    R3,#+5` |
| `}` | `MOVS    R4,#+6` |
| | `MOVS    R5,#+7` |
| | `MOVS    R6,#+8` |
| | `MOVS    R7,#+9` |
| | `POP     {R0,R4-R7}` |

## 4.5.4 Data Types

### 4.5.4.1 *Methods and Parameter type*

The ARM architecture is RISC load/store, so before applying any operations on variables, they must first be loaded into registers. In 4.2.2 it was shown that the ARM Cortex-M0+ microcontroller has 32bit wide general-purpose registers. As a consequence before applying any operations to registers which contain 8/16bit variables, they first have to be extended to a word. To do that the compiler applies `UXT(H/B)` assembly instruction. The following example, illustrates this process (code is compiled on optimization level – **low**). The compiler adds two extra instructions to extend the registers.

16bit variables C code

```
unsigned short int foo(unsigned short int
a,unsigned short int b)
{
 return a+b;
}
int main()
{
 int res=foo(1,2);
 return res;
}
```

16bit variables assembly code

```
foo:
UXTH     R0,R0
UXTH     R1,R1
ADDS     R0,R0,R1
BX       LR
main:
PUSH     {R7,LR}
MOVS     R1,#+2
MOVS     R0,#+1
BL       foo
POP      {R1,PC}
```

If the parameters and return type of the method were 32bit wide, it wouldn't have been necessary to extend the registers. As a result in that case it is much more efficient to use unsigned integers instead of smaller data types.

#### *4.5.4.2 Local variables*

The ARM Cortex M0+ microcontroller can efficiently load and store 8bit, 16bit and 32 bit variables. The Thumb instruction set can handle word-extending and register loading into a single assembly instruction – for example LDRB can be used to extend and load an 8bit variable into the 32bit registers. The flexibility of the instruction set is shown in the following example, where variables of type `char`, `short int` and `char` are loaded into registers by using a single instruction.

16bit variables C code

```
int a;
char b;
unsigned short int c;
int main()
{
 unsigned short int a1=a;
 unsigned short int b1=b;
 unsigned short int c1=c;
 return a+b+c;
}
```

16bit variables assembly code

```
main:
LDR      R0,??main_0
LDR      R1,[R0, #+4]
LDRB     R2,[R0, #+0]
ADDS     R1,R1,R2
LDRH     R0,[R0, #+2]
ADDS     R0,R1,R0
BX       LR
```

### 4.5.5 Multiple Load(s) & Store(s)

Earlier it was observed that, before applying any operations on variables they must first be loaded into registers. As a consequence, a program developed for a RISC architecture, will have to use a significant number of load (LDR) and store (STR) instructions. The ARM Cortex M0+ microcontroller does however support instructions for multiple loads (**LDM**) and multiple stores (**STM**).

The following example illustrates, that it is possible to combine five load (`LDR`) assembly instructions into a single one. The gain in code size is more than obvious.

| Using multiple load instructions | Using a single instruction for multiple loads |
|---|---|
| `LDR   r1, [r0], #4` | `LDM r0!, {r1,r2,r3,r4,r5};` |
| `LDR   r2, [r0], #4` | |
| `LDR   r3, [r0], #4` | |
| `LDR   r4, [r0], #4` | |
| `LDR   r5, [r0], #4` | |

Analogically it is possible to combine multiple store (STR) instructions into a single one. Again the reduction in code size is significant.

| Using multiple store instructions | Using a single instruction for multiple stores |
|---|---|
| `STM   r1, [r0], #4` | `STM r0!, {r1,r2,r3,r4,r5};` |
| `STM   r2, [r0], #4` | |
| `STM   r3, [r0], #4` | |
| `STM   r4, [r0], #4` | |
| `STM   r5, [r0], #4` | |

Using multiple loads and stores can significantly improve the code density of a program. It does however make the register allocation more complex and might often lead to performance penalty.

### 4.5.6 Global Variables

#### *4.5.6.1 Accessing Global Variables*

In C language global variables are variables which can be accessed from outside of the scope of a method. Global variables are usually stored in stack (in-case not constant), thus they are never directly allocated to registers. As a consequence the compiler must first generate an instruction, which loads the address of the variable into a register. The following example illustrates this process:

| Global variable access – C code | Global variable access – assembly code |
|---|---|
| `int a;` | `MOVS      R0,#+5` |
| `int main()` | `LDR       R1,??main_0 //Load adr of a into reg1` |
| `{` | `STR       R0,[R1, #+0]` |
| `  a =5;` | `MOVS      R0,#+0` |
| `}` | `BX        LR` |

To conclude in the general case accessing global variables is more expensive than using local ones. The programmer should aim to minimize the usage of global variables. When those variables are only being used within the same module they should be declared as static which will allow further optimizations.

#### *4.5.6.2 Initializing Global Variables to Zero*

According to the C Standard [REF] (static) global variables are guaranteed to get initialized to zero, by default on any compiler. As a result there is no need for the programmer to manually initialize them. This would only add unnecessary overhead- a single STR instruction for each initialization, plus a MOV one for copying zero to a register. The following example illustrates the unnecessary overhead, caused by the zero initialization.

| Global variable init to zero– assembly code | Global variable init to zero– assembly code |
|---|---|
| `int a;` | `LDR      R0,??main_0` |
| `int main()` | `MOVS     R1,#+0` |
| `{` | `STR      R1,[R0, #+0]` |
| `  a =0;` | `STR      R1,[R0, #+4]` |
| `}` | `MOVS     R0,#+0` |
|  | `BX       LR` |

### 4.5.7 Cache Locality

Locality is a term being used, when referring to the same value, or related storage locations being frequently accessed in the same order (e.g. arr[0], arr[1], arr[2]). To improve (spatial) cache locality one can group consequently accessed variables, inside a structure- the compiler will be able to use a single pointer to access them. In addition members being closed to each other in the memory layout, may open further possibilities for compiler optimizations.

### 4.5.8 Bitfields

Bit-fields are very poorly specified within the C Standard. The compiler chooses how bits are allocated within the bit-field container. When using bit-fields the compiler will often add extra shifting and masking instructions, which will result in extra overhead. To conclude the programmer should avoid using bit-fields, if not completely required.

## 4.6 RSA- ACHIEVING HIGH CODE DENSITY

This chapter discusses the possible optimizations presented in chapter 4.5 , in respect to their applicability and impact to the Reusable Software Architecture repository. In addition a recommendation on the significance of the optimization technique is made and whether the effort required to apply them is justifiable. Finally advice on the optimal in terms of code size compiler setup is given.

### 4.6.1 Data Structures Memory Layout- Optimization Applicability

#### *4.6.1.1 Current Situation*

Chapter 4.5.1 explained that the compiler aligns the member of a structure by adding padding between them. The RSA contains a decent amount of structures. However as seen in 4.4.3 according to the current implementation, mostly small data types are used. Moreover variables of type `char` can be aligned on any address. Often structure members are already ordered, thus no significant amount of extra padding is present.

#### *4.6.1.2 Code Density Impact*

Considering the statements made above, there are still some possibilities for small reductions. Table 5 gives a list of some of the structures where the extra padding was removed. It is possible that there are other cases in which structures can be optimized, however for the moment such were not found.

| **Structure:** | **Potential reduction (bytes):** |
|---|---|
| **TS_TIMER** | 4 |
| **TS_QUEUE** | 2 |
| **TS_AUTO_BACKUP** | 8 |
| **TS_DALIVARS** | ? |

**Table 5**

#### *Conclusion & Recommendation*

To conclude the RSA, does not at the moment have a significant memory overhead caused by structure padding. The reduction is negligible, thus the effort spend on the issue is probably unjustifiable. If the RSA goes to using larger datatypes, it is very likely that the topic might have to be reconsidered.

### 4.6.2 Unaligned Memory Access- Optimization Applicability

#### *4.6.2.1 Current Situation*

Chapter 4.2.3 explained that the ARM Cortex M0+ microcontroller does not support unaligned memory access. As consequence the compiler has to use extra instructions to ensure the alignment of the variables.

Inside the RSA, some structures where unaligned memory is forced by applying the `#pragma pack(pop)` compiler directive are present. As a result accessing those structures causes extra code.

#### *4.6.2.2 Code Density Impact*

Three cases of "packed structures" were identified within the RSA. In the first case structures from the dalislave_dc module, are being accessed frequently. As a result the caused code overhead is significant.

In the second case all members of the structures inside dalimb_dc module are of type `char` (can start on any address), thus no code overhead is present.

In the last case the structures inside submb are being accessed frequently, which leads to code size overhead. Table 6, gives a summary of the potential code size reduction across the three modules

| **Structures in module:** | **Potential reduction (bytes):** |
|---|---|
| **dalislave_dc** | 544 |
| **dalimb_dc** | 0 |
| **submb** | 156 |

**Table 6**

#### *4.6.2.3 Conclusion & Recommendation*

To conclude unaligned memory access, forced by packed structures within the RSA, causes a significant amount of extra code memory. It is recommended that the `#pragma pack(pop)` directive is removed. In case it is actually required that a structure must packed, such compiler directives should be applied at the latest possible stage (e.g. before storing a structure to EEPROM). Otherwise, the more a packed structure is being accessed, the more extra instructions will be added by the compiler.

### 4.6.3 Function Register Allocation- Optimization Applicability

#### *4.6.3.1 Current Situation*

Chapter 4.5.3 explained that if the programmer aims to write methods with a maximum of 4 parameters and 8 local variables, no stack memory will be used and all variables will reside in registers. Almost all methods inside the RSA, fulfill the above mentioned requirement.

#### *4.6.3.2 Code Density Impact*

Methods within the RSA, do not overuse local variables nor take a big amount of parameters. As a result there isn't a significant stack usage caused by spilled variables.

#### 4.6.3.3 *Conclusion & Recommendation*

No effort is required for optimizing the RSA methods. However it is still beneficial, that programmers are aware with this optimization concept.

### 4.6.4 Data Types- Optimization Applicability

#### 4.6.4.1 *Current Situation*

Chapter 4.5.4 explained that before applying any operations to registers which contain 8/16bit variables, they first have to be extended to a word, thus the compiler adds zero-extending assembly instructions. As noted in chapter 4.4.3 the RSA was initially implemented for an 8bit microcontroller. As a consequence mostly 8/16bit variables are being used for both parameters and method type.

#### 4.6.4.2 *Code Density Impact*

The ARM Cortex M0+ efficiently loads 8/16bit into registers with LDRB and LDRH instructions. When variables are parameters or return types **UXTH** and **UXTB** instructions are used. Table 7 shows the number of zero-extend instructions used in the RSA (compiled with optimization level – high for code size). The two instructions come from the Thumb set, thus they both occupy 2 bits of memory. The potential code size reduction is also calculated below.

| **Instruction:** | **Count:** | **Potential reduction (bytes):** |
|---|---|---|
| **UXTH** | 47 | 94 |
| **UXTB** | 109 | 218 |

Table 7

#### 4.6.4.3 *Conclusion & Recommendation*

To conclude using 32bit variables for parameters and method return type will bring decent code size reduction. Local variables are at the moment efficiently loaded into registers in terms of code size, because the microcontroller can load and zero-extend a variable into registers with a single assembly instruction.

### 4.6.5 Multiple Load(s) & Store(s) - Optimization Applicability

#### 4.6.5.1 *Current Situation*

Chapter 4.5.5 explained that the ARM Cortex M0+ microcontroller, supports instructions for loading and storing multiple variables into registers. As shown in Table 8 load and store in structure within the RSA, take a significant amount of the code memory. It was found that the compiler, does almost never use multiple load and store instructions.

| **Instruction:** | **Count:** |
|---|---|
| **LDR(H/B)** | 1954 |
| **STR(H/B)** | 1173 |

Table 8

#### *4.6.5.2 Conclusion & Recommendation*

To conclude the code produced by the compiler is possibly not optimal in terms of code size. However at the moment the only way to solve this issue is by rewriting parts of the code in assembly. Due to the nature of the RSA this is not preferable. It is recommended that the compiler support is contacted and the reason for not using multiple load/store instruction is found.

### 4.6.6 Global Variables- Optimization Applicability

#### *4.6.6.1 Current Situation*

Chapter 4.5.6 explained that it is more expensive to access global variables than local variables. Global variables are not commonly used within the RSA- all global variables which are "shared" within the **same** module are also declared static. The compiler can efficiently optimize static variables. However, unnecessary initialization of global variables to zero, was found.

#### *4.6.6.2 Code Density Impact*

Inside module swdalidrv global (static) variables are being initialized to zero. Removing the statements will bring some small code size reduction.

| **Module:** | **Potential reduction (bytes):** |
|---|---|
| **swdalidrv** | 28 |

Table 9

#### *4.6.6.3 Conclusion & Recommendation*

To conclude there is no need to take extra effort in optimizing global variables within the RSA. It is still beneficial that programmers are aware, that it is not required to initialize global (static) variables to zero.

### 4.6.7 Cache Locality- Optimization Applicability

#### *4.6.7.1 Current Situation*

Chapter 4.5.7 explained the principle of cache locality- the same value, or related storage locations being frequently accessed in the same order. In that sense, cache locality optimizations can be applied to frequently accessed large structures.

#### *4.6.7.2 Code Density Impact*

One of the largest and most accessed structures within the RSA is the TS_DALIVARS. It is being used from more than 5 separate modules. Having more than 20 members it is also the largest structure in the repository. Reordering its members in a sequence they are being accessed brings a notable code size reduction. The potential code reduction benefits are shown in Table 10.

| **Structure:** | **Potential reduction (bytes):** |
|---|---|
| **TS_DALIVARS** | 68 |

Table 10

#### 4.6.7.3 Conclusion & Recommendation

To conclude optimizing the order of structure members as for cache locality brings decent code size reduction. It is however hard to apply, as all modules who access the structure have to be examined, and the optimal sequence have to be determined.

### 4.6.8 Bitfields- Optimization Applicability

#### 4.6.8.1 Current Situation

Chapter 4.5.8 explained that the compiler has to add extra assembly instructions when accessing bitfields. Within the RSA there is only one structures which has bitfields as members.

#### 4.6.8.2 Code Density Impact

The TS_CONFIG_INFO structure from the dalivariables module is the only place where bitfields are used within the RSA. As shown in Table 11 removing the bitfields will bring 52 bytes of code reduction.

| Structure: | Potential reduction (bytes): |
|---|---|
| **TS_CONFIG_INFO** | 52 |

Table 11

#### 4.6.8.3 Conclusion & Recommendation

To conclude bitfields are not commonly used within RSA – the only place where such are found should be optimized and none should be used in the future.

### 4.6.9 Optimization Techniques- Results Summary

In the previous chapters it was proved that some of the optimizations are applicable to the RSA and can bring code reduction. As shown in Table 12 the total potential code reduction is **1056** bytes, which is **5.6%** of the RSA modules code.

| Total potential reduction (bytes): | Total potential reduction (%) |
|---|---|
| **1056** | 5.6 |

Table 12

### 4.6.10 The Compiler- Conclusion & Recommendation

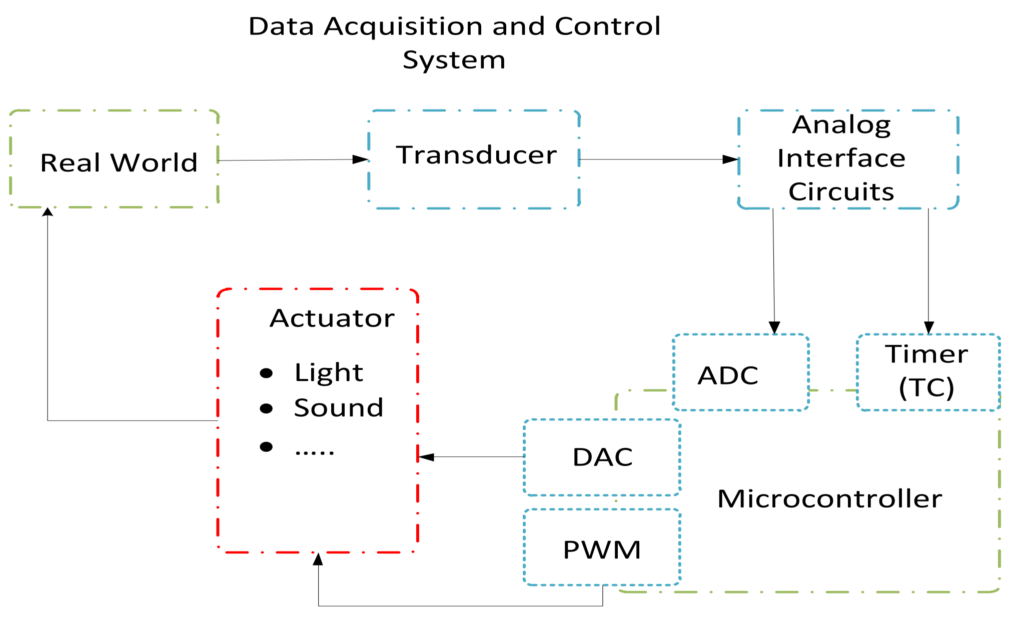
The currently being used compiler – EWARM is considered to be a good choice. In chapter 4.3.5.4, it was proved that there is no significant difference between EWARM and the other available compilers. It was found that the compiler already reduces the program code size by 30%, producing a very high-dense code No further investigation on the compiler is required.

# 5 ADC Sampling with DMA & EVSYS

The following chapter give an overview of the second part of the assignment- investigating the usage of Direct Memory Access (DMA) and Atmel Event System (EVSYS) to reduce the CPU overhead caused by ADC sampling.

## 5.1 PROBLEM DOMAIN

In real life we have analog signals such as light, sound, electricity and others. Those real-world signals have to be converted into digital one, before they can be understood and manipulated by digital equipment, such as microcontrollers. This translation process is called ADC sampling and is done using Analog-to-Digital Converter (ADC). ADC sampling is often mandatory for embedded systems.



Graph 17

LED drivers are no exception to this statement. To serve they purpose, they have to continuously perform analog-to-digital conversion sampling. As a result a significant part of the CPU resources are being reserved for this. In addition while conversion results are being accessed, no other software actions can be executed.

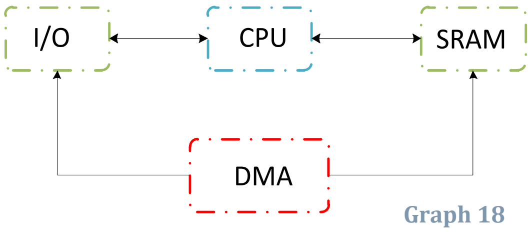
The SAM D 21 microcontroller, has two hardware blocks– DMA and EVSYS, which provide direct peripheral to peripheral communication. It is possible, that in combination with each other, they might be used to complete the ADC sampling process without any CPU intervention.

## 5.2 DIRECT MEMORY ACCESS

### 5.2.1 DMA Introduction

Direct memory access (DMA) is a controller feature that enables high rate data transfer. DMA is used in various hardware system such as graphic cards, network cards, sound cards and disk drive controllers. Data transfer can be executed from:

- Memory to memory.
- Peripheral to memory.
- Memory to peripheral.
- Peripheral to peripheral.



Graph 18

### 5.2.2 DMA Features

The DMA controller can, without any utilization of the CPU, transfer data between peripherals and memory. The data transferred is referred to as transactions. There are several ways to trigger a new DMA transfer:

- Software trigger can be used to start DMA transfer (e.g. the programmer can do that from code).
- Peripheral trigger can be used to start DMA transfer (e.g. ADC peripheral can invoke new transfer).
- Atmel Event System generated event can be used to start a DMA transfer.

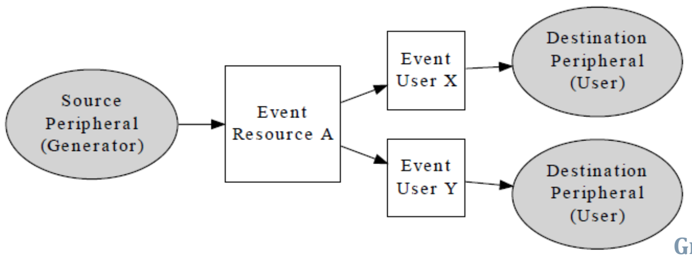
## 5.3 ATMEL EVENT SYSTEM

### 5.3.1 Atmel Event System Introduction

Atmel Event System (EVSYS) allows autonomous, low-latency configurable communication between the SAM D 21 peripherals. Contrary to traditional interrupt-based systems, communication is executed without any CPU utilization or consuming resources such as bus or RAM bandwidth.

### 5.3.2 Atmel Event System Features

On the SAM D 21 microcontroller, peripherals can be configured to produce and receive event signals. Peripherals which receive events are called event users and peripherals which produce events are called event generators.


Source Peripheral (Generator)
Event Resource A
Event User X
Event User Y
Destination Peripheral (User)
Destination Peripheral (User)

Graph 19

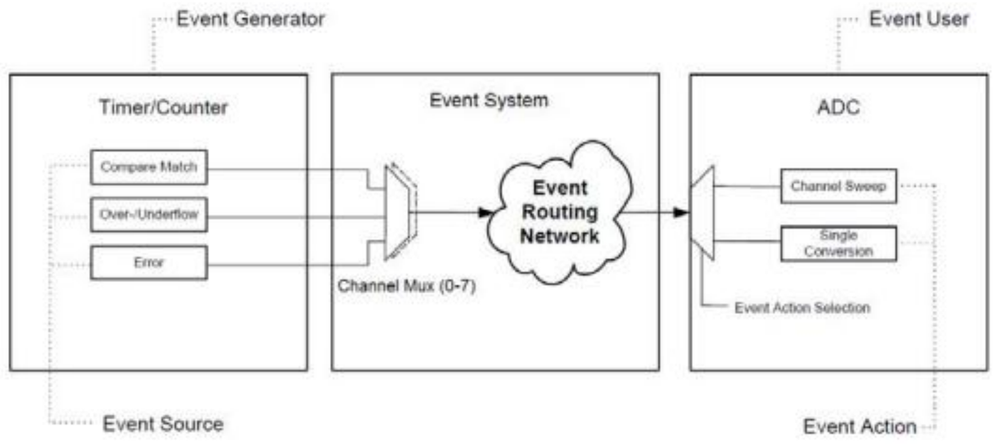
The EVSYS also defines event resources, which can be configured to select the input peripheral that will generate the event signal, the synchronization path and the edge detection method. Simply put event resources serve as a link between the event generator and event users.

## 5.4 DMA & EVSYS

In chapter 5.1 it was explained that the microcontroller often has to reserve part of the CPU resources for ADC sampling. In attempt to avoid this issue DMA and EVSYS can be used- both modules do not require any CPU utilization.

In an embedded system ADC samples are taken on a specified time frame, called sample rate. For this purpose a Timer (TC driver) is used- on a known interval, interrupt is raised, which invokes the new ADC conversion. The interrupt blocks the ongoing process and calls the ADC driver. The EVSYS can be used to replace this interrupt based concept.

Graph 20 depicts an example where a timer peripheral (TC) is used as an EVSYS event generator, to trigger a new ADC conversion. As a result the initial interrupt trigger is avoided,which is already an improvement.



Graph 20

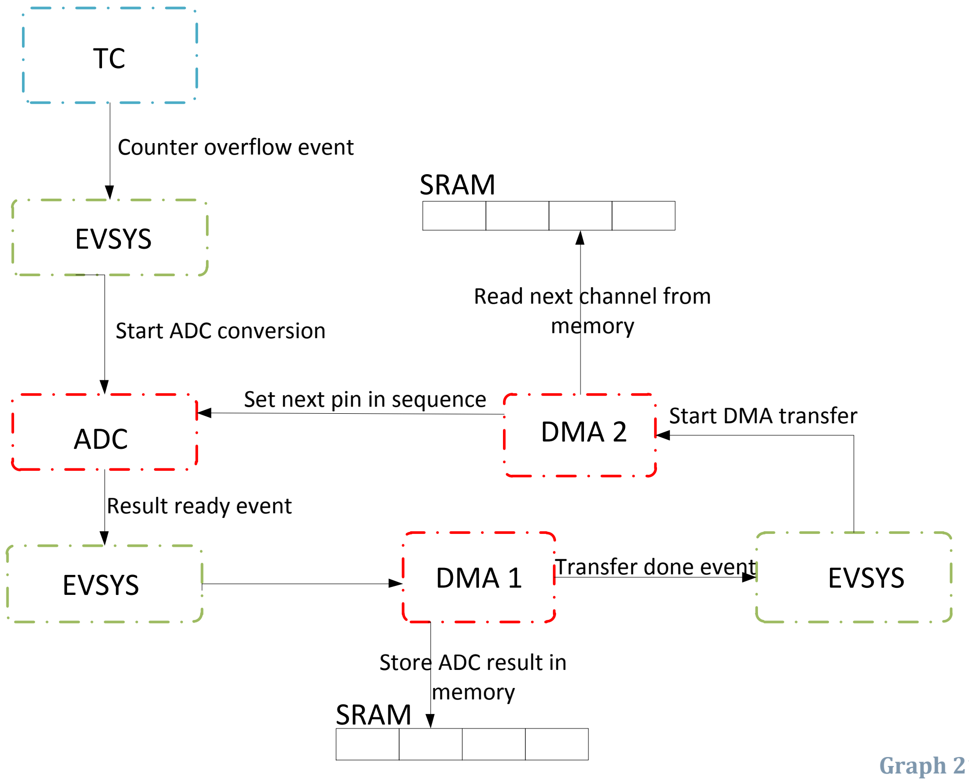
For the moment however, the CPU must still be used to transfer the results from the ADC to memory. In chapter 5.2 the usage of DMA for memory transfer, without intercepting the CPU was explained. In addition, it was noted that a DMA transfer can be triggered by an event signal from the EVSYS, which makes DMA applicable to the current solution.

There is as final improvement which can be made- currently the ADC can sample only a single channel which is initially being configured. It is possible to set the channels via DMA. For the purpose a second DMA transfer needs to be executed, which reads the next channel to be sampled from memory and configures the ADC.

The final members of the solution are as follows:

- TC (timer) which generates event signal on a specified interval of time.
- EVSYS resource #1 which routes the generated event signal to the specified peripheral.
- ADC which receives the event from resource #1, performs analog-to-digital-conversions and generates an event signal when done.
- EVSYS resource #2 which routes the generated event to the appropriate peripheral.
- DMA #1 which receives the event from resource #2, reads the ADC sample result, translates it into memory and generates an event signal when completed.
- EVSYS resource #3 which routes the generated event to the appropriate peripheral.
- DMA #2 which receives the event from resource #3, reads the next channel to be sampled from memory and configures the next channel in the ADC driver.

The whole process is presented in Graph 21.



Graph 21

The discussed proposal to use DMA and EVSYS in combination, improves the current situation by removing the overhead of the continuous ADC sampling from the CPU. In addition, the whole process can be executed parallel to the normal on-going tasks, which will improve the performance of the LED driver

# 6 Conclusion(s) and Recommendation(s)

**The first** part of the graduation project covered different means and techniques for achieving high code density on the ARM Cortex M0+ microcontroller. The research was focused on the three main factors which are important for the topic- the microcontroller, the compiler and the source code. As a result of this study a number of possible optimizations were defined and analyzed. Their applicability to the Reusable Software Architecture was determined and the possible memory reduction was measured.

The final results showed that there are some possibilities for code reduction. In addition it was also proved, that the compiler does already apply a significant amount of optimizations reducing **30%** of the program. This only leaves a few improvement possibilities to the programmer which could bring a potential code memory reduction of **5.6%**.

Based on the outcome of the project it is not believed that further investigation is required in the light of manual code size optimization techniques. The topic of code density might however be researched from the hardware restrictions (microcontroller) point of view- it might be possible that a specific microcontroller architecture can achieve a more dense code the ARM Cortex M0+ and still be cost competitive.

**The second** part of the graduation project carried an investigation on the concept of reducing CPU usage by redistributing some of the software functionality to hardware components. In the specific case the study was focused on DMAC and Atmel EVSYS.

During the execution of the research it was found that it might be beneficial to use DMA and Atmel EVSYS in combination, to implement flexible ADC Sampling without any CPU utilization. Demo application was created as a proof of concept, which showed that it is indeed possible to use both hardware components for the given problem domain.

Further investigation can be done in detecting other aspects in which DMA and Atmel EVSYS can be used. It might be interesting to research whether DMA can be used in certain cases to reduce the load and store operations executed by the CPU and in that sense lead to increased code density.

## *Evaluation*

The graduation project was without any doubt an interesting and challenging experience. A lot of new knowledge in the field of embedded software was obtained. The project gave me a realistic view, of what the stage of my current knowledge and skills is, giving me the chance to improve certain aspects of my professional profile. It also brought me insight and motivation to continue my future career in embedded software development.

# References

1. Andrew N.Soss, D. S. (2004). *ARM System Developer's Guide.* Morgan Kaufmann .
2. *ARM Cortex M0+ Technical Reference Manual.* (n.d.). Retrieved from http://infocenter.arm.com/help/topic/com.arm.doc.ddi0484c/DDI0484C_cortex_m0p_r0p1_trm.pdf
3. *ARM Software Development Toolkit.* (1998). Retrieved from http://infocenter.arm.com/help/topic/com.arm.doc.dui0041c/DUI0041C.pdf
4. *Atmel SAM D 21 Datasheet.* (n.d.). Retrieved from http://www.atmel.com/images/atmel-42181-sam-d21_datasheet.pdf
5. *Atmel SAM D21 Xplained Pro- User Guide.* (n.d.). Retrieved from http://www.atmel.com/images/atmel-42220-samd21-xplained-pro_user-guide.pdf
6. *IAR C/C++ Development Guide.* (2009, June). Retrieved from http://supp.iar.com/FilesPublic/UPDINFO/004916/arm/doc/EWARM_DevelopmentGuide.ENU.pdf
7. Oshana, R. (n.d.). *Software Engineering of Embedded and Real-Time Systems.* Elsevier.
8. *Procedure Call Standard for the ARM Architecture.* (2012, November 30). Retrieved from http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E_aapcs.pdf
9. Valvano, J. W. (2014). *Embedded Systems: Introduction to Arm® Cortex(TM)-M Microcontrollers.*
10. Yiu, J. (2011). *The Definitive Guide to the Arm Cortex-M0.* Newnes.
11. *GNU Wiki*. Retrieved from https://gcc.gnu.org/wiki

# Appendix A

Machine-dependent optimizations are those applied with internal knowledge of the target machine architecture. The compiler will take advantage of microcontroller specific information for the instruction set and memory layout (registers, stack, heap etc.) to produce more efficient code.

## ❖ PEEPHOLE OPTIMIZATION

Peephole is a compiler optimization technique which involves examining a set of assembly instructions and when possible replacing this set of instructions with such that it is shorter or executes faster. There are hundreds of different peephole optimizations. Example of peephole optimization is presented below, where the **fourth** assembly instruction - `LDR R0` is a redundant load and can be removed.

| C source Code | Not optimized Assembly Code | Optimized Assembly Code |
|---|---|---|
| `a = b + c;` | `LDR  R0, b      //R0=b` | `LDR  R0, b      //R0=b` |
| `d = a + e;` | `ADD R0, R0, c //R0=R0+c` | `ADD R0, R0, c //R0=R0+c` |
| | `STR  a, R0     //a=R0` | `STR  a, R0     //a=R0` |
| | `LDR  R0, a      //R0=a` | `ADD R0, R0, e //r0=R0+e` |
| | `ADD R0, R0, e //r0=R0+e` | `STR  d, R0     //d=R0` |
| | `STR  d, R0     //d=R0` | |

## ❖ INSTRUCTION SCHEDULING OPTIMIZATION

Instruction scheduling is a compiler performance optimization technique which involves reordering the machine-code instructions in a way that minimizes the total number of cycles required to execute a specific sequence. There are various techniques for instruction scheduling, one of the most popular **loop unrolling** is presented below, where the loop overhead is reduced by replicating the body of the loop twice. While bringing performance loop unrolling has a negative effect on code size.

| Not optimized C code | Loop fusion applied on C code |
|---|---|
| `for (i = 0; i < 100; i++)` | `for (i = 0; i < 50; i++)` |
| `  foo();` | `{` |
| | `  foo();` |
| | `  foo();` |
| | `}` |

## ❖ REGISTER ALLOCATION OPTIMIZATION

Register allocation is an optimization technique where the compiler analyzes the function to determine which values should be assigned to the registers at each moment of the function execution. For example the compiler would often allocate frequently used variables (loop counters etc.) to registers. If there are not enough registers available, the compiler chooses which register to “spill” in memory.

# Appendix B

Machine-independent optimizations are those which the compiler can perform regardless the target processor/microcontroller. Such techniques will usually try to improve the flow of the code. Examples of common machine-independent optimizations are:

## ❖ HOISTING

Hoisting is a compiler optimization technique which involves moving loop-invariant code outside of the loop body, without affecting the meaning of the program. Example of hoisting is presented below where the `y + z` and `x * x` statements are irrelevant to the loop structure.

| Not optimized C code | Hoisting applied on C code |
|---|---|
| `for (int i = 0; i < n; i++) {`<br>`    x = y + z;`<br>`    a[i] = 1337 * i + x * x;`<br>`}` | `x = y + z;`<br>`t1 = x * x;`<br>`for (int i = 0; i < n; i++) {`<br>`    a[i] = 1337 * i + t1;`<br>`}` |

## ❖ COMMON SUBEXPRESSION ELIMINATION

Common subexpression elimination is a compiler optimization technique which involves searching for expression which evaluate the same value and analyzing whether it is feasible to replace them with a single variable holding the computed value. Example of Common subexpression elimination is presented below where the `b * c` statement is producing the same value both times.

| Not optimized C code | Subexpression elimination applied on C code |
|---|---|
| `a = b * c + g;`<br>`d = b * c * e;` | `tmp = b * c;`<br>`a = tmp + g;`<br>`d = tmp * e;` |

## ❖ DEAD CODE ELIMINATION

Dead code elimination is a compiler optimization technique which involves identifying sections in the source code of a program the result of which is never being used. If not removed those section are still being executed leading to waste of computation time and memory resources. Example of dead code elimination is presented below where the `int result = a/b;` statement result is never used.

Not optimized C code

```c
int foo (int a, int b)
{
    int result = a/b;

    return a*b;
}
```

Dead code elimination applied on C code

```c
int foo (int a, int b)
{
    return a*b;
}
```

## LOOP FUSION

Loop fusion is a compiler optimization technique which involves identifying loop structures which share identical statements and merging them into a single one. Example of loop fusion is presented below where the `for (i = 0; i < n; i++)` statement result is identical for both loops.

Not optimized C code

```c
for (i = 0; i < n; i++)
  a[i] = a[i] + n;

for (i = 0; i < n; i++)
  b[i] = b[i] + n;
```

Loop fusion applied on C code

```c
for (i = 0; i < n; i++)
{
  a[i] = a[i] + n;
  b[i] = b[i] + n;
}
```

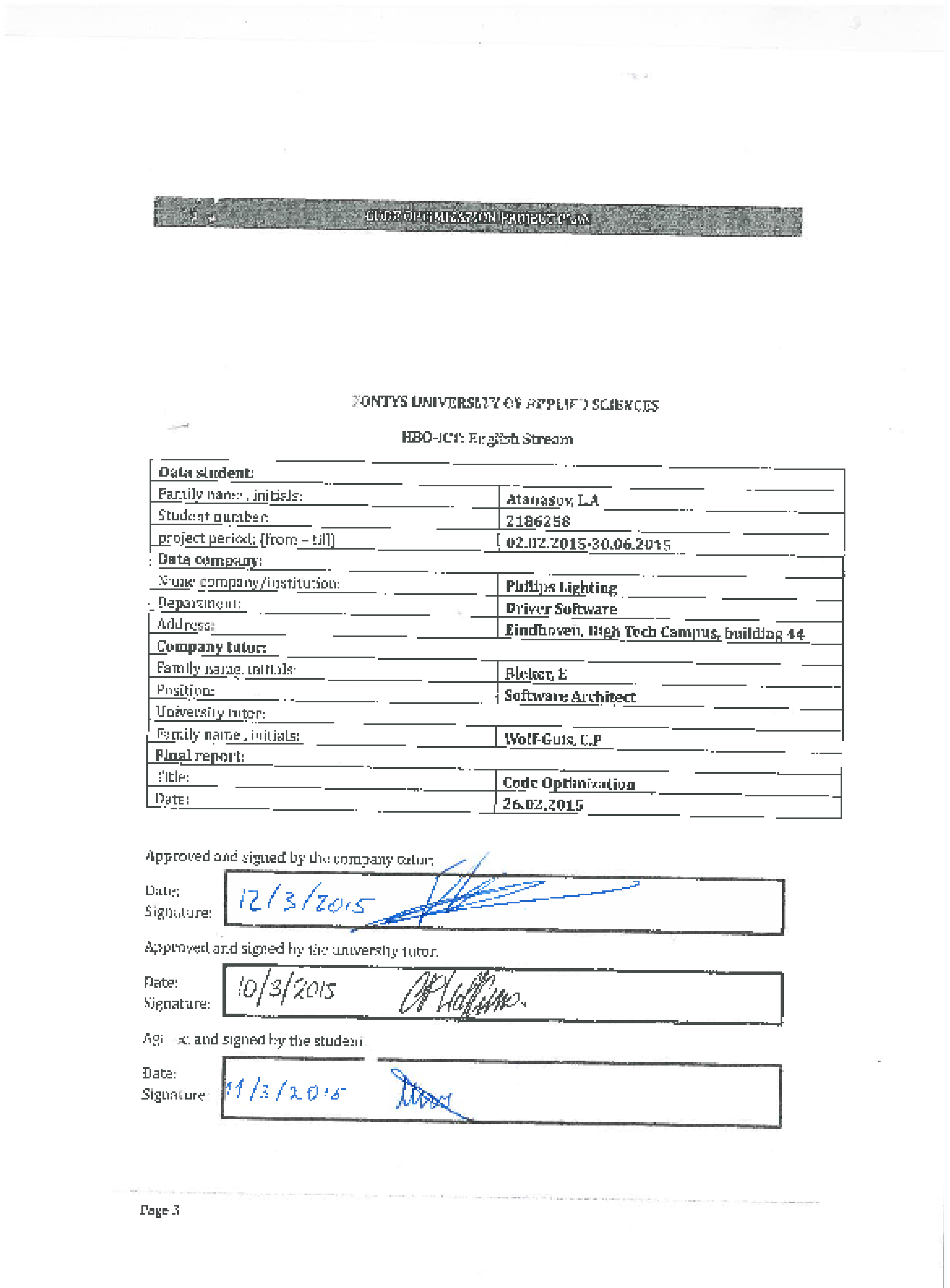
# Appendix C

# CODE OPTIMIZATION PROJECT PLAN

FY 2015

*Philips Lighting graduation project*

## Contents



CODE OPTIMIZATION PROJECT PLAN

FONTYS UNIVERSITY OF APPLIED SCIENCES

HBO-ICT English Stream

| | |
|---|---|
| **Data student:** | |
| Family name, initials: | **Atanasov, L.A** |
| Student number: | **2186258** |
| project period: (from – till) | **02.02.2015-30.06.2015** |
| **Data company:** | |
| Name company/institution: | **Philips Lighting** |
| Department: | **Driver Software** |
| Address: | **Eindhoven, High Tech Campus, building 44** |
| **Company tutor:** | |
| Family name, initials: | **Bleker, E** |
| Position: | **Software Architect** |
| University tutor: | |
| Family name, initials: | **Wolf-Guts, C.P** |
| **Final report:** | |
| Title: | **Code Optimization** |
| Date: | **26.02.2015** |

Approved and signed by the company tutor:

Date: 12/3/2015

Signature: [illegible]

Approved and signed by the university tutor:

Date: 10/3/2015

Signature: [illegible]

Agreed and signed by the student:

Date: 11/3/2015

Signature: [illegible]

Page 3

# Project Statement

## FORMAL CLIENT

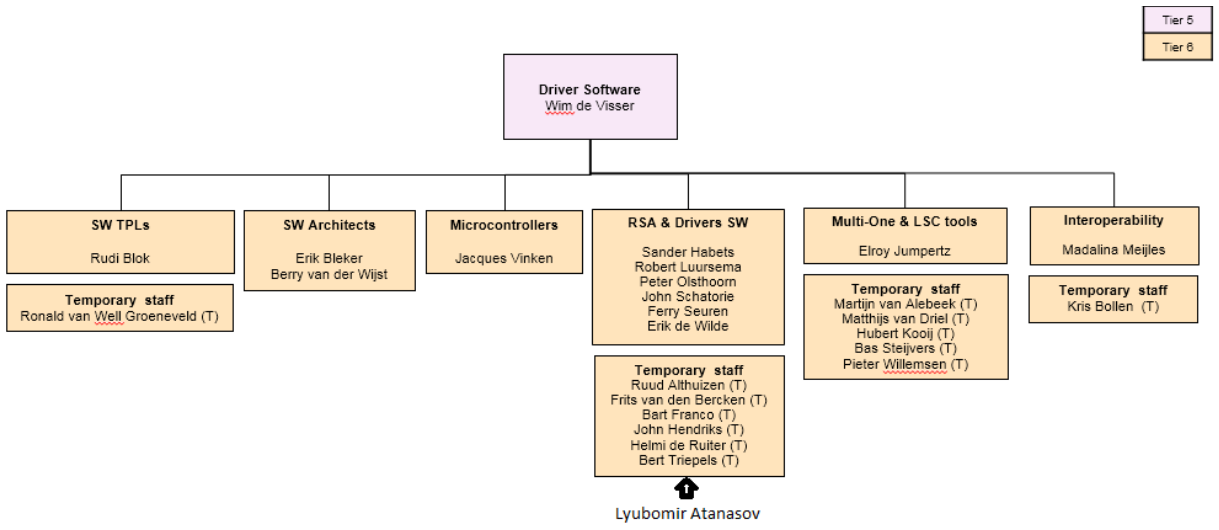
Mr. Erik Bleker, who is in the positon of Software Architect within the RSA team will act as the formal client during the project execution.

## PROJECT LEADER

The Graduate Intern – Lyubomir Atanasov, will be responsible for the successful execution of the project plan. Mr. Rudi Blok who is in the position of Project Leader in the RSA team will assist in keeping track of the project progress, execution and planning of the SCRUM Sprints.

## ORGANIZATION STRUCTURE

During the execution of the project the graduate intern - Lyubomir Atanasov will be part of the RSA team. The organizational structure is depicted on the chart below.



*Organizational chart (**confidential**)*

## GENERAL OVERVIEW

Philips Lighting is specialized in offering professional lighting solutions for the business. The company manufacture products which are often hosts of an embedded system.

Microcontrollers, which are microcomputers incorporating the processor, RAM, ROM and I/O ports into a single package, are often employed in an embedded system. There is a vast variety of different microcontroller vendors available on the market such as:

- ARM
- Atmel
- Intel
- NXP Semiconductors
- STM

It is also common for vendors to sell microcontrollers based on different architectures.

## INITIAL SITUATION

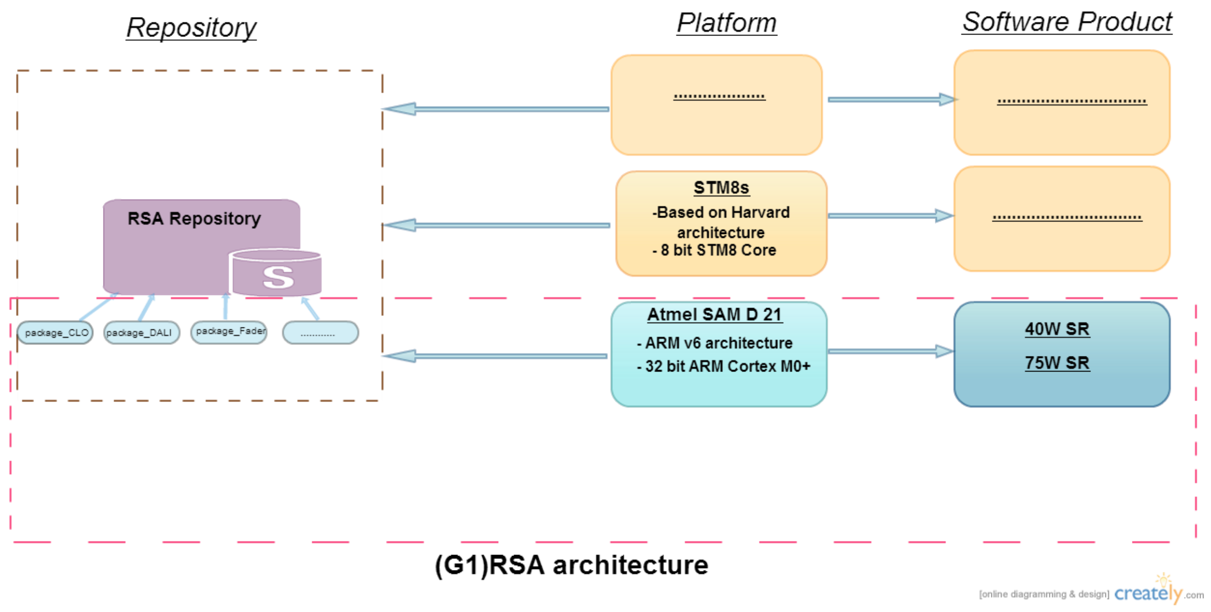
The Driver Software department at Philips Lightning is responsible for developing software solutions for different hardware products. LED Electronic drivers are one of the lightning solution products for which software is being developed. LED drivers are devices which can convert incoming AC power to the proper DC voltage, and control the current flowing through the LED. They can be used to provide dimming, color-changing and other capabilities.

Drivers might have different characteristics and requirements, thus they would sometimes have microcontrollers based on a different platform and architecture (ARM, Atmel, Intel, STM etc.).

Platforms would however often share similar or even same functionality. As a result common functionality is separated into modules/packages and stored into a repository named *RSA (Reusable Software Architecture)*. The *RSA* repository is being maintained and developed by the RSA team.

Projects for different platforms can re-use packages from the *RSA* repository by applying minor configurations. This process is depicted on Graph *G1.*

Previously the STM8S 8-bit platform was being used for developing software projects. Recently, the Atmel SAM D21 platform was introduced and the existing code was ported for it. Despite the ported software can be executed, it is not fully optimal as the two platforms have certain possible differences.



(G1)RSA architecture

## PROJECT OBJECTIVE

The goal of the project is to investigate the possible means to optimize the code for the Atmel SAM D21 platform. The scope of the project is depicted on Graph *G1* with a dotted red line. Optimization will be applied in the following aspects:

### Code Optimization

Code size/density usually refers to the number of assembly instructions needed to perform a requested action and the amount of space each instruction takes. The less space an instruction takes and the more work per instruction it performs, the higher achieved density of the code is. Code size might directly affect the available memory and performance of the software.

### Direct Memory Access Controller

The Direct Memory Access Controller (DMAC) provides fast and CPU independent data transfer between the memories and the peripherals. The result is high speed communication with minimal CPU load. The DMAC adds data bandwidth, increases performance, off-loads the CPU and saves power.

### Atmel Event System

The Event System provides inter-peripheral signaling without CPU or DMA usage. No CPU cycles or interrupts are needed. As a result no overhead is present and one peripheral is able to signal and trigger actions in other peripherals when something happens. The eventual advantage is less CPU load and current consumption.

## PROJECT JUSTIFICATION

The ported code was previously designed for a platform with different technical characteristics. As a result it might be producing unnecessary memory usage thus taking extra space. Optimizing the code for the current platform might lead to increase speed, smaller binaries and allow adding additional functionality in the future without having to upgrade the hardware. As it is not needed to add more physical memory, there are no additional money costs added to the final product.

## PROJECT DELIVERABLES

The Project deliverables are specified according to the three main optimization aspects. They are not directly correlated to each other, but will all accumulate some benefit to the code.

### Code Optimization Related

#### *Research Document*

Research Document covering the possible Code size optimizations must be delivered. The document must contain overview of the discovered optimization techniques and comparison between the optimizations which different compilers apply. Compulsory appendices to the Research Document are:

- ❖ Test Plan document describing how and which techniques will be evaluated. The test plan will also define a baseline for comparison to which the impact of the applied optimizations will be measured.
- ❖ Code repository containing the performed evaluation tests.

#### *Code analysis report*

The current code should be studied. Parts of the code, patterns, and structures which might be a subject of optimization based on the Research Document output, must be outlined in the report.

#### *Code Optimization*

The code optimization techniques which are found to be feasible during the research are to be applied to the code specified in the code analysis report.

Applied optimizations must be documented, including result measurements of the actual benefits compared to the previous code.

### Direct Memory Access Controller Related

Investigate the possibility of using DMA to transfer data from ADC channel to the memory and deliver:

- ❖ General overview document of DMA and how it can be used for the mentioned purpose.
- ❖ Demo application.

### Atmel Event System Related

Investigate the possibility of using the event system with the Timer Counter Control (TCC) and deliver.

- ❖ General overview document of Atmel Event System and how it can be used with TCC and other peripherals.
- ❖ Demo application.

## PROJECT CRITERIA OF SUCCESS

The assignment is to be considered successful when the outcome of the project:

- ❖ Provides a well-documented research, from which reasonable deductions and recommendations can be made.
- ❖ Brings insight in the possibilities of code optimization applicable to the *RSA* repository and the specific platform.
- ❖ Brings insight in the possible usages of DMA and Atmel Event System for the specific context.

## PROJECT NON-DELIVERABLES

The client would not expect as a delivery:

- ❖ Completely optimized RSA repository which can replace the existing one for the on-going projects.
- ❖ DMAC usage and TCC Event System incorporated inside the existing projects.

## PROJECT CONSTRAINTS

There are several aspects which have to be considered while applying optimization of the existent code. The target platform is Atmel SAM D21 which would imply the use of:

- ❖ ARMv6 architecture
- ❖ Thumb-2 instruction set

The overall design of and structure of the RSA repository must not change and the way a platform can reference to it must remain the same. Code optimization should be applied but the readability of the code must not suffer. On-going work on the same project by other software engineers should also be considered.

## PROJECT RISKS

| Risk | Likelihood | Impact |
|---|---|---|
| High level of technical complexity | Medium | High |
| Misunderstanding of deliverables | High | High |
| Poor project planning | Low | High |
| Inadequate estimation of project scope and time | Medium | High |
| Ineffective communications | Low | Medium |

# Plan of Execution

## PROJECT METHODOLOGY

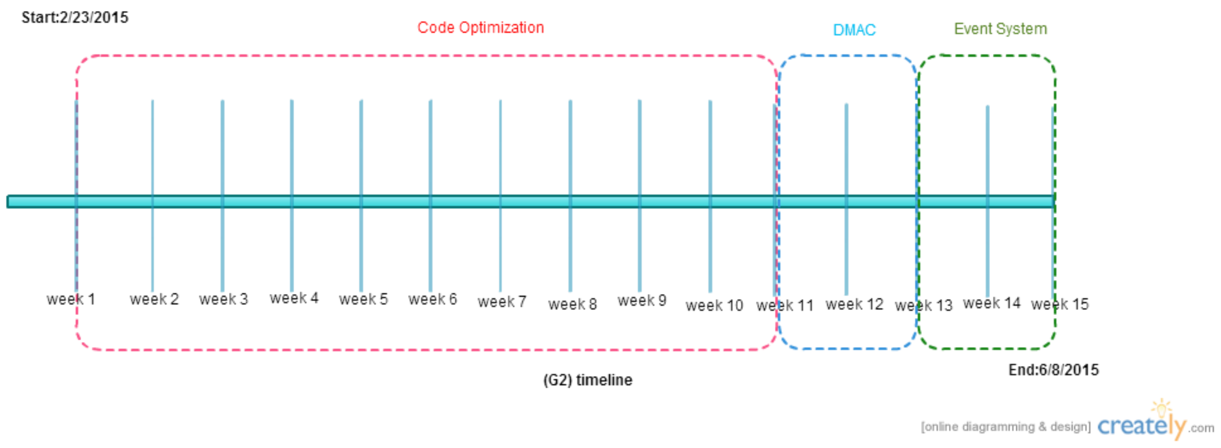
The project will follow an agile/scrum development methodology while incorporating the DOT Research framework. When defining new tasks and activities, the strategy/strategies to which they relate will also be noted.

## PROJECT PHASING

The project will be executed according to the time line depicted on Graph *G2*. The project will take 105 days and will be separated in 3 sprints of 3 weeks and 3 sprints of 2 weeks. Each sprint will have number of tasks which cover one or more of the strategies defined in the DOT framework.

- ❖ The Code Optimization part of the assignment will take **3** (3w) sprints and **1**(2w) sprint.
- ❖ The DMAC investigation part of the assignment will take **1** (2w) sprint.
- ❖ The Atmel Event System part of the assignment will take **1** (2w) sprint.

The first sprint will start on the 23th of February and the project will end on the 8th of June.



## References

- http://www.atmel.com/Images/Atmel-42181-SAM-D21_Datasheet.pdf
- http://dkc1.digikey.com/ca/en/tod/Atmel/XMEGA/XMEGA.swf
- http://eee.guc.edu.eg/Courses/Electronics/ELCT912%20Advanced%20Embedded%20Systems/Lectures/ARM%20System%20Developer%27s%20Guide.pdf