Bachelor's thesis

Degree programme in Information Technology

Embedded systems

2017

Oskari Teeri

IMPLEMENTING A FAN CONTROLLER

TURKU AMK

TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR’S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Degree programme in Information Technology | Embedded systems

2017 | 32

Oskari Teeri

IMPLEMENTING A FAN CONTROLLER

The purpose of thesis was to study and implement a fan controller which is controlled from the USB port. Data structures and possible implementation methods were studied, particularly for embedded systems.

The platform used in the thesis was Microchips AVR microcontroller, and its open source libraries. ATMega32u4 was chosen for the microcontroller. The software was created during the thesis. The software controls of the fan is based on user commands.

The LUFA was chosen for the USB control. The microcontroller receives user commands via a USB-CDC protocol, which was available through the LUFA library.

The thesis studies data structures in theory, and their suitability in embedded systems. Additionally, modern C++ is compared to the currently implemented C program. Finally, the thesis discusses the suitability of the C++ for embedded platforms in general.

KEYWORDS:

AVR, C, C++, linked list

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietotekniikka | Sulautetut ohjelmistot

2017 | 32

Oskari Teeri

TUULETINOHJAIMEN TOTEUTUS

Tämän opinnäytetyön tarkoituksena oli toteuttaa yleinen – sulautettuihin järjestelmiin soveltuva projekti, käyttäen helposti saatavilla olevia avoimia ohjelmakirjastoja ja osia. Työ ei ollut toimeksianto, vaan se toteutettiin itsenäisenä projektina. Opinnäytetyössä toteutettiin tuuletinohjain, jota ohjattiin USB-väylän kautta annettavilla komennoilla. Tietorakenteita ja mahdollisia toteustapoja tutkittiin erityisesti sulautettuihin järjestelmiin soveltuvalla tavalla.

Alustana käytettiin Microchipin AVR-mikrokontrolleria, ja sille tarkoitettuja avoimen lähdekoodin kirjastoja. ATMega32u4 valitiin mikrokontrolleriksi, sen sisältämien ominaisuuksien vuoksi, ja sille toteutettiin opinnäytetyön aikana ohjelma, jonka tarkoitus oli ohjata useita tuulettimia käyttäjän komentojen perusteella.

Ohjelman USB-ohjaukseen käytetään LUFA-kirjastoa, joka on AVR-alustalle tarkoitettu avoin USB-ohjelmakirjasto. Mikrokontrolleri vastaanottaa käyttäjän komennot USB-CDC-protokollan kautta, jonka toteutus oli tehty LUFAssa. Lisäksi opinnäytetyössä tehtiin USB-ohjelmakirjastojen välistä vertailua.

Opinnäytetyössä tutkittiin teoriatasolla tietorakenteita, ja niiden yleistä soveltuvuutta sulautettuihin järjestelmiin. Erityistä huomiota kiinnitettiin linkitettyjen listojen toteutukseen. Työ selvitti sulautetuissa järjestelmissä käytettäviä C- ja C++-standardeja, sekä niiden eroavaisuutta tavallisen ohjelmistokehityksen näkökulmasta. Lisäksi vertailtiin modernia C++:aa ohjelman nykyiseen, C-kieliseen toteutukseen. Lopulta tutkittiin C++:an yleistä soveltuvuutta sulautettuihin järjestelmiin. Tähän kuului esimerkiksi C++:an turvallisuutta heikentävien ominaisuuksien tutkiminen reaaliaikasysteemeissä.

C++-osuutta ei toteutettu työn aikana, vaan se perustui teoreettiseen pohdiskeluun. Käytännössä työn aikana toteutettiin ainoastaan toimiva C-kielinen ohjelma. Työn tuloksia voisi soveltaa samankaltaista C++-projektia suunniteltaessa.

ASIASANAT:

AVR, C, C++, linkitetty lista

CONTENT

ALGORITHMS

APPENDICES

FIGURES

TABLES

LIST OF ABBREVIATIONS (OR) SYMBOLS

ASF	Atmel Software Framework
BoM	Bill of Materials
CR	Carriage Return
DRC	Design Rule Check
EDA	Electronic Design Automation
IDE	Integrated Development Environment
ISP	In-System Programming
IEEE	Institute of Electrical and Electronics Engineers
LF	Line Feed
LUFA	Lightweight USB Framework for AVR
MOSFET	Metal Oxide Semiconductor Field Effect Transistor
OOP	Object Oriented Programming
POSIX	Portable Operating System Interface
PWM	Pulse Width Modulation
RTOS	Real Time Operating System
RTTI	Run-Time Type Information
SCM	Source Control Management
STL	Standard Template Library
USB	Universal Serial Bus
USB-CDC	Universal Serial Bus Communications Device Class
UML	Unified Modeling Language
UNIX	Uniplexed Information Computing System
QoS	Quality of Service

1 INTRODUCTION

The thesis tries to solve the problem of fan speed adjustment in different systems. There is no universal standard of controlling Pulse Width Modulation (PWM) fans in desktop computers. It is up to the operating system to provide the fan control capabilities. Despite the fact that the thesis focused on controlling fans, the result could have been applied elsewhere where PWM controlling is required.

The fan controller could have been inserted straight in to the motherboards USB pins, or it could be controlled from external USB ports. The idea of the fan controller was that it could have been assembled inside the desktop computer. The power of the fan control is drawn from the USB port.

There are commercial fan controllers available, where the control is exerted by tuning the potentiometers, but not through USB. Some operating systems have capabilities for controlling fan speeds, but they are not guaranteed.

The user controls the fan speeds by sending commands via a terminal that is being transmitted through USB-CDC protocol. The USB-CDC is meant for emulating old terminals as a data transfer method.

2 REQUIREMENTS

This chapter defines the project requirements, both software and hardware. Requirements for the user interface and availability, such as command line and support in various systems, are covered in this chapter.

2.1 Hardware

The need for embedded application comes from the surrounding physical world and usually the software requirements are derived from the hardware requirements and not vice versa.

There could be other defining factors for e.g.: price, schedule, standards, ethics and so on. These factors are not essential in the sense of making a working prototype, but are helpful in other areas.

The design principle for this project was to keep the hardware layout as simple as possible, using the smallest amount of available hardware components. Using fewer components in a design has its advantages; there are fewer spare parts to be worried about in the future.

If there is a possibility to do hardware feature in software, it is worth implementing it in software. This drastically reduces the amount of components being used. It also makes the design cheaper, at least in theory.

The fans should be PWM controllable; not all fans are. PWM controlled fans can be recognized from their pin count. Fans with three pins are usually PWM controllable, and fans with four pins have usually feedback for automatically updating their speed.

2.2 Software

The fan controller should be controlled from the command line. The commands should be close as possible to a natural language. The communication protocol should be available for the most popular operating systems.

The USB protocol was chosen for controlling the fans, due to its universal support in almost any system. The goal was to be able to control the fans, despite of the underlying system. Given the criteria, there were no better alternatives for communication between the host and the device, other than USB.

The fan controller should be interfaced through a USB-CDC protocol. The CDC stands for communications device class. Additional drivers might be needed, depending on the host operating system. The USB-CDC drivers should be available on the most popular operating systems.

2.2.1 Control commands

As the fans are controlled by different PWM channels, there needs to be an update method for each of the PWM channel value. If the user wants to set a fan to a certain speed, the correct register value needs to be updated – by scaling it – based on the given per cent value. There needs to be an abstraction between the fan speed percentage and the actual register value, since the user cannot remember all the register values for different fan speeds.

There should be a way to set arbitrary number of fan speeds within a single command. If every fan update required a different command, updating all the fans would be inefficient and slow. Implementation needs to have a mechanism for updating all the fans in one command.

The command structure should be `command target value`, where the `target` argument flags fans urging for an update. The `target` argument should accept `all` target flag for updating all the values at once.

The `value` argument is represented as a fan speed percentage from 0–100. The value argument should accept `low`, `med`, `high`, `default` and `auto` keywords. The `auto` keyword would be an optional argument, meant for future implementations, if the fans ever set their own speed based on temperature.

The `target` argument should accept `all` keyword for updating all of the values at once.

Valid example commands, by the given definition:

- `set fan0: 30` – set fan 0 to 30 % PWM duty cycle
- `set all: 50` – set all fans to 50 % PWM duty cycle
- `set all: default` – set all fans to default % PWM duty cycle

There needs to be a method for setting multiple targets and their values at once. The command line should accept updating different targets separated by commas.

Valid example command, with commas added:

- `set fan0: 30, fan1: 50` – set fan 0 to 30 % and fan 1 to 50 % PWM duty cycle

Adding *comma* as a control character adds great flexibility for giving many arguments at once. The command line syntax is now close to the given requirement: natural language and self-explanatory.

3 IMPLEMENTATION

This chapter describes how the program was implemented, and how the development environment was configured. The implementation is reflected, and improvements considered. The generic nature of the implementation is also discussed.

3.1 Program flow

The program has two C array buffers for catching the user input. The first buffer is initialized for the ring buffer, which is implemented in LUFA. The second buffer is initialized for the parser, which was implemented during the thesis.

The main program loop waits for CR and LF characters to appear in the data stream. If the line end characters are detected, the second buffer is given to the parser. The parser then tokenizes the strings in to chunks and then pushes them in to a queue in the right execution order.

If the tokenized string is an available command and it is found from the function table, then the function is going to be executed. If the tokenized string is not found from the function table, the user will be notified by sending the *malformed input* message. The notification is sent by printing the `help` command at the end user.

The program flow is described in the Figure 1 as a UML sequence diagram. It covers key functionality of the modules from the perspective of the main loop after the line end characters are detected.

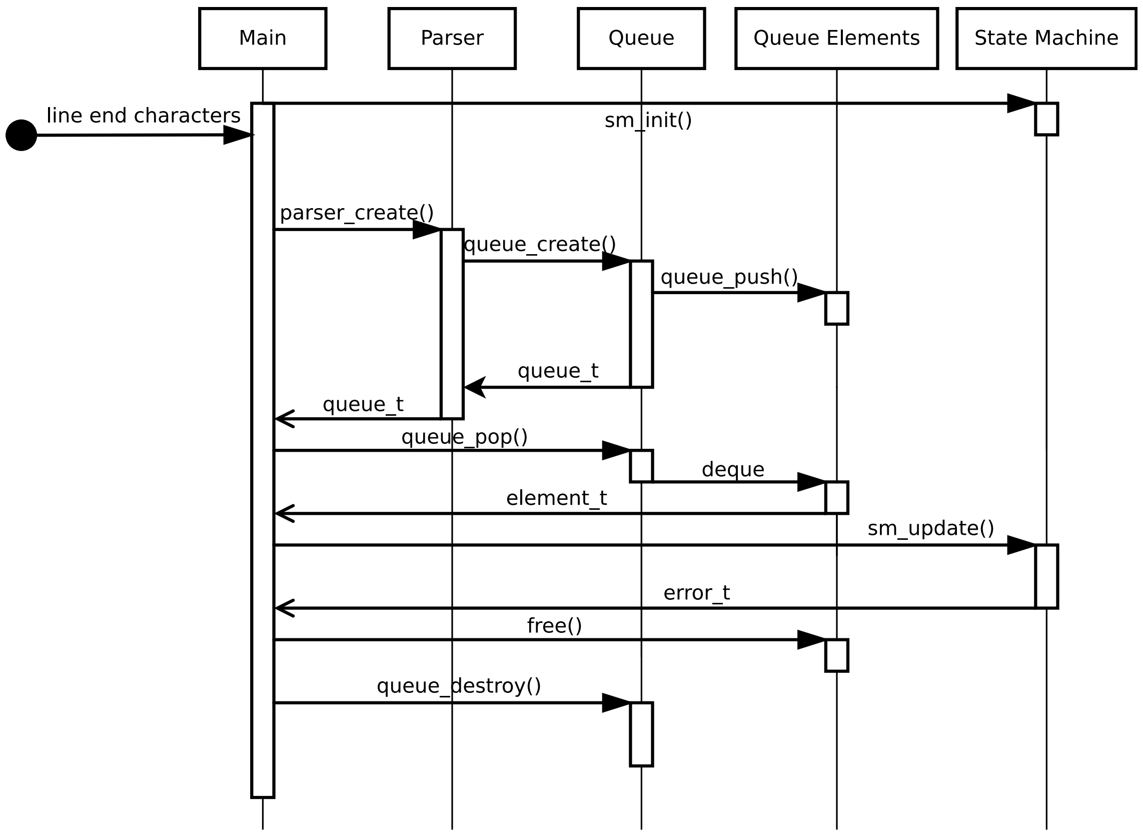


Figure 1. UML sequence diagram of the program functionality.

The queue elements are also in the sequence diagram for helping to understand memory allocation semantics. The main loop is responsible of freeing the allocated memory elements from the queue. After the user input is handled, the allocated queue nodes are being freed with the `queue_destroy` function.

3.2 Parser

The parser uses `srtok_r` function for tokenization of the user input. The main string is split by searching a special set of characters known as control characters, then doing a new recursive search with a new set of control characters by nesting the earlier search. This is repeated until all the "tokens" are collected. In this context, token means a control character; space, comma, colon or command. Control characters are used for separating commands.

The parsing does not work with `strtok`, since it uses internal global variables, which are by definition non-recursive and non-reentrant. This was the reason `strtok` was replaced with `strtok_r`. `Strtok_r` is a IEEE Std 1003.1-1988 ("POSIX.1") extension for the standard C and is implemented in the AVR's GNU C library called avr-libc. [1]

POSIX extensions are common, but the design goal should always be in standard C. This is important, especially when the application requires portable and reliable code, which is a common requirement for embedded software design. Shipped production code should always be written in standard C. The parser was first implemented using `strtok`, but then replaced with `strtok_r`, due to lack of capability in doing recursive calls.

The parser could be implemented in other ways. One possible implementation could have been input formatting with `scanf`, which is a "reverse operation" of the function `printf`. This option was not studied during the thesis.

3.3 Queue

The queue is implemented as a doubly linked list. Nodes are added or removed as the queue size changes (Figure 2). The arrows represent pointers, and the blocks represent the queue elements.

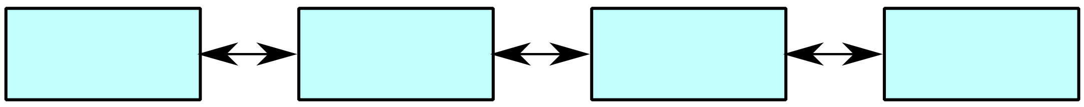


Figure 2. The data structure of the queue is a doubly linked list.

Nodes of the queue and element data are allocated from the heap by a memory allocation function called `malloc`.

The parsed commands and control characters are pushed in to the queue, despite the fact that they might be invalid. Each queue element is searched from the function table, and if they are found; the corresponding function is going to be executed. This is known as dispatch table.

If the user has given invalid data, the function is not found from the dispatch table, and the program execution jumps to an error routine, which could be informing the user about the malformed user input.

3.4 State machine

The state machine finds the correct function by iterating over the strings. The right function is executed by its name. If the name is not found, then the command string is not executed. A function can only be executed in a state where it is defined, since the state machine limits all functions by their defined state. The function must be defined in a state where it is being executed.

Valid commands will be executed if they are in a valid sequence and the state allows it. In some case, even control characters might be executed. In the thesis, control characters were only used for state changes. It is possible to bind any function to any command or control character.

The state machine implementation has three valid states; `CMD`, `TARGET` and `VALUE`. The valid commands can only be given in an order that fulfills the previously mentioned state transition order. Valid commands could go through one, two or three states, but they need to transition them in the right order. For example, the command `help` will go through only one state; `CMD`, where it prints the help text for the user and finishes there.

Table 1 explains what state changes have been gone through by the valid example command "`set fan1: LOW, fan2: MED`".

Table 1. State transitions of "`set: fan1: LOW, fan2: MED`".

Input		set	:	,	:	
State	CMD	TARGET	VALUE	TARGET	VALUE	CMD

As the Table 1 shows, every command starts and returns to a state called `CMD`. This is the initial state, where state machine starts parsing the user input. The input arguments `fan1`, `LOW`, `fan2`, and `MED` does not affect the state transitions, which is why they are not shown in the Table 1. Only invalid input argument could affect the state transitions. In the current implementation, invalid input simply returns to an initial `CMD` state.

The state machine includes a `fan_controller` header file, which controls the fans. This means the `fan_controller` has all the local data required for calculating the PWM register values from per cent values. The `fan_t` structs are stored in an array as instances, and accessed by their indexes.

3.5 Generic

Generic programming is a sub category of programming paradigm, where the goal is to reduce the code duplication. It is an abstract concept, which aims for reusable code in different use cases. Generic programming has different meanings, depending on the context. For e.g.: between C and C++.

The function table implementation could be viewed as a generic implementation. It is also known as a branch table, where the function pointers are stored in an array. The array consists of struct pointers, which are constructed for mapping between the C strings, and their corresponding function pointers.

3.6 Improvements

The existing program could be improved in different ways. This sub chapter introduces possibly better approaches for implementing the program.

3.6.1 More generic

The parser control characters; comma, colon and space are statically implemented, meaning they are not configurable by the programmer (without re-implementing the parser). This is not a huge drawback, since the control characters are unlikely to change between different implementation iterations.

This does not mean the programmer cannot change the state machine functionality. The statically implemented part only concerns the control characters; comma, colon, space and the logic behind of them being parsed.

The parser should be implemented as more generic, where everything is user configurable.

3.6.2 Memory allocations

The queue nodes and its elements were dynamically allocated by `malloc`. This should be avoided in embedded programs, due to the nature of memory being fragmented over time. Executing `malloc` might take varying amount of time, or it might even fail. Considering the embedded applications, memory fragmentation is not a good thing, since embedded programs could have years of uptime.

Hardware

The schematics were designed in Eagle. Eagle is electronic design automation (EDA) software, meant for designing printed circuit boards and laying out the components.

Eagle was chosen for the thesis due to its popularity, free price and large component libraries. Some manufacturers offer Eagle design rule check (DRC) file – a set of guidelines meant for catching the errors made in the design phase.

The fans are driven by MOSFETs. The parts were collected from http://mouser.com, with the bill of material (BoM) importer tool. All of the parts were listed in the Appendix 1. The parts were chosen based on their existing library implementations.

4 AVR

This chapter gives general overview of the Atmel AVR ATmega32u4 microcontroller, which was chosen for controlling the fans. It covers Arduino, bootloaders and USB stack. Arduino is a development environment meant for hobbyists, but was used during the thesis for prototype development. The software stack of Arduino Leonardo was not used.

The main reason ATmega32u4 was chosen is because it had a built-in USB hardware, USB software stack and 7 PWM channels [2]. It is possible to use components such as hardware PWM multiplexers, but as the earlier 'simple design' rule states: use as few components as possible.

4.1 Bootloader

Every official Arduino product comes with a pre-installed Arduino bootloader. Application code can be uploaded in to the Arduino only by using the Arduino IDE, unless another bootloader or programmer is being used.

Arduino libraries are not suitable for industrial applications, as they are meant for educational purposes only.

The ATmega32u4 flash memory has a separate section for the bootloader. It could be write-protected, which prevents the bootloader flash section being overwritten while the user application is being uploaded. The bootloader firmware can only be burnt if the correct fuse bytes are being set. [2]

The flash section, reserved for the bootloader, can be overwritten with application code if there is no need for advanced features. With this approach, maximal flash memory for the user applications can be achieved. If there is no bootloader, the application code must be uploaded by the ISP pins. [2]

The flash section size of the bootloader can be changed by programming the fuse bytes. The bootloader program size must be less than the reserved bootloader size. [2]

4.2 USB 2.0

The USB 2.0 standard supports three types of throughputs [3]:

- low-speed (10 – 100 kb/s)
- full-speed (500 kb/s – 10 Mb/s)
- high-speed (25 – 400 Mb/s)

The ATmega32u4 USB implementation supports only low and full speed operations. The USB 2.0 standard has two operating modes; device and host. Both host and device applications are available in the ATmega32u4.

4.2.1 ASF

Atmel provides Atmel Software Framework for reducing application development time. The ASF consists of ready-made software modules, which can be added to any project inside the Atmel Studio IDE.

ASF software modules can be added inside the Atmel Studio IDE through ASF wizard, which automatically includes necessary header files as module drivers. It is also possible to download ASF as a standalone package if the development environment is other than the Atmel Studio. ASF is available for GCC and IAR compilers.

The ASF has a software module for USB 2.0 drivers. Unfortunately, it does not support the ATmega32u4, which belongs to the Atmel AVR8 microcontroller series. The USB driver support is only available for XMEGA and the AVR32 microcontroller families.

Writing an USB driver for the ASF library is possible, but this option was not examined during the thesis. The ASF was not chosen for the project, because it did not include full support over the USB stack.

4.2.2 LUFA

LUFA (Lightweight USB Framework for AVRs) is an open source implementation of the AVR's USB stack, licensed under the permissive MIT. Commercial license is also available. It features open source bootloaders, demo USB projects, both device- and host applications. There are lots of demo class driver implementations based on different use cases.

The LUFA package contains open source bootloaders for different use cases.

5 DATA STRUCTURES

This chapter discusses on abstract data structures, and how they were implemented during the thesis. This chapter could be viewed as a general introduction to data structures, and why they were chosen for the particular implementation.

5.1 Ring buffer

Ring buffer is a first in, first out (FIFO) abstract data type, derived from the singly linked list (Figure 3).

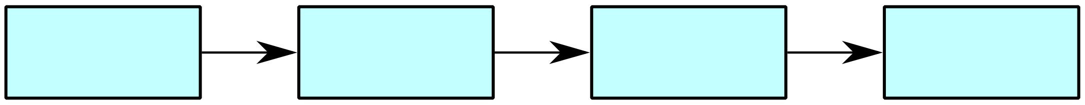


Figure 3. Singly linked list.

The ring buffer is an optimal implementation solution for reading data streams in the embedded systems. The difference between a singly linked list and a ring buffer is that the ring buffer has no real end node (Figure 4).

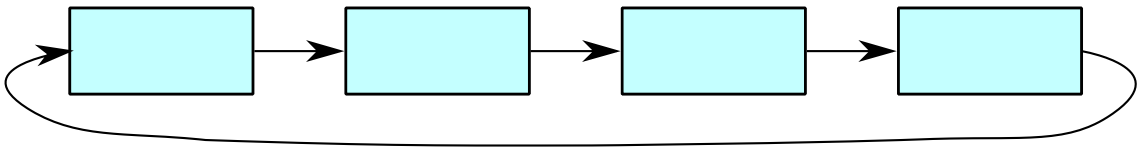


Figure 4. Ring buffer.

Operations are constant time $O(1)$, and they could be implemented as a fixed size length for achieving maximum performance [4]. This also means ring buffers cannot overflow.

The data is read from the USB port; byte by byte (or char by char in this case) and pushed to the ring buffer as it arrives. The ring buffer implementation comes from LUFA. The downside of using ring buffers is the need of second buffer for collecting the characters, since C standard string manipulation functions only support manipulating strings as arrays, where the memory layout is contiguous; every memory address is next to each other.

If C standard string manipulation functions supported singly linked lists or custom iterators, this would not be the case. If the data arrives in a reversed order, extra buffer might be needed for flipping the data.

One improvement could be implementing C standard string manipulation functions as singly linked lists, so there would not be a need for extra buffers. C++ Standard Template Library (STL) has the option of implementing custom iterators for every container type, unlike C standard library, which only supports standard C arrays as an input.

5.1.1 Drawbacks

Linked lists are not optimal when the *n*th element of the data structure is accessed directly, since all the nodes need to be traversed (in the worst case scenario), which makes the operation $O(n)$, where n is the traverse count [4]. In data streams, this is not the common case, since the data needs to be accessed in a sequential order; from front or tail – depending on the need.

There are many misguided ring buffer implementations available on the Internet [4]. The most common mistake is implementing the ring buffer as a standard C array, and then using modulo arithmetic for accessing its elements. This is demonstrated in Algorithm 1.

Algorithm 1. Implementation with modulo arithmetics.

```
buffer[index % BUFFER_SIZE];
```

In some implementations [4], modulo operation is used for preventing out of bound access of the data structure. Modulo operations should not be used in performance sensitive application. It makes the linked list implementation extremely slow, especially for embedded platforms, where the instruction set is limited and compiler cannot optimize the code properly.

In addition, the array access method of using modulo operation is also commonly implemented with a bitwise `AND` operation (Algorithm 2), which optimizes modulo operation, if and only if the size of the buffer is a natural number that is in power of 2. [4]

Algorithm 2. Implementation with modulo arithmetics and bitwise `AND`.

```
buffer[index & (BUFFER_SIZE - 1)];
```

This optimization limits the possible buffer sizes to 2nd complement numbers [4], which is unnecessary, because modulo operation in data stream buffers could be achieved as varying fixed sized, by linking the first and the last node together.

The linked list implementation for modulo operation should only be implemented if the data structure elements needs to be accessed in a sequential order, one by one. Data streams are usually accessed one by one in a sequential order, so they could be implemented as circular singly linked lists.

5.2 Queue

Queue is an abstract last in, first out (LIFO) data type, where doubly linked list implementation is an optimal solution, since both directions need to be traversed.

The queue was implemented by dynamically allocating the node data for them. The elements were also allocated dynamically, since their sizes were not known before the tokenization of the user input string.

6 DEVELOPMENT

This chapter introduces the development environment. The development environment consists of project files, source control management and short description of testing.

6.1 Project files

The project files were organized in the following directory structure:

- lufa
- sch
- src
- tests

The *lufa* directory contains necessary parts of the LUFA, which are required by the compilation process. This directory contents should not be touched, unless the USB software stack implementation itself needs to be modified.

The *sch* directory is reserved for schematics included in the thesis.

The *src* directory contains essential parts of the project, including key functionality and business logic. It also contains USB descriptor files, configuration files, makefiles and any other project specific files.

The *tests* directory is reserved for separately testing independent parts of the software. Embedded platforms are not optimal for testing purposes, since their input/output accessibility is often limited, which is why the tests were designed to run also on a desktop machine. The separation between business logic and hardware dependent code eases the process of writing more portable code. The tests itself were written in C, and should work in any platform with a C compiler.

6.2 SCM

The LUFA project is included as a git sub module, which is an optimal solution for embedded projects, where the main project and the sub project requires isolation; changes made in the sub module have no effect inside the main module.

This is useful if the sub module is rarely updated and its maintenance work is done by someone else. Therefore, sub module is an optimal solution for inclusion of sub projects, which are not updated frequently. Sub projects could be: RTOS'es, USB software stacks, etc.

6.3 Testing

Each of the portable software parts were tested in a desktop environment before flashing them in to the microcontroller. This eased the development process and produced portable code by default. Program modules, using dynamic memory allocations such as queues, were tested against memory leaks with valgrind. Valgrind is an UNIX program for memory leak detection and profiling.

Measuring embedded code performance should be avoided in a desktop environment, since the results might differentiate between different architectural systems [5]. Modern processors with caching might produce unexpected results, compared to simple architectures which usually do not cache memory.

Cheap and simple architectures, such as embedded 8-bit architectures, without memory management unit do not have caching abilities. The performance difference is most notable when the algorithm handles the input as a contiguous memory, such as array, where caching becomes possible. Algorithms, such as singly linked lists, which handle data via an indirect memory access, such as pointers, are not cache-friendly [5].

7 C++

This chapter gives common overview of general techniques for programming with C++ in embedded systems. The scope of this chapter is the thesis, but overall embedded development is considered as well. Thought process of this chapter started as solving the problems in the thesis, but then extended to thinking the theory between C and C++. The thesis was entirely implemented in C, which makes the comparison between C and C++ is purely theoretical.

7.1 Standard

It is important to mention which standard we are referring to. C++ has had major changes over the years. The newest official standard is C++14, which has minor changes and fixes compared to C++11, which was considered to be the greatest change in the history of C++ [6].

Usually the compiler support drags behind new standards, especially in the embedded platforms, where the new compiler support needs to be separately implemented. It is unrealistic to assume that all the features from new standard are immediately implemented as they arrive. The implementation lag has been reduced over the years, but is still a valid concern.

An embedded developer needs to consider the risks every time a new software piece is released; it is likely to contain more bugs, or even unsafe features, which will be discovered in the future. Maturity of the software matters. Jumping to a new standard for an embedded developer might be a risk.

For e.g. a new C++11 memory management feature, called `std::auto_ptr`, was later deprecated in the C++14 standard [6]. The feature was discovered useless – using `std::shared_ptr` and `std::unique_ptr` was recommended instead. _LÄHDE_

7.2 Differences to C

C++ is a programming language based on classes, or objects. It is also known as Object Oriented Programming (OOP). The biggest difference between C and C++ is the mindset, since they are semantically very similar languages, but in practice are used very differently [6].

The main difference between C and C++ is how they interact with memory. In C++, the objects are first constructed with a constructor and then automatically destructed via a destructor when the object exits its scope.

In C, the memory allocation and freeing could be done anywhere in the program – unlike in C++, where the ownership is usually much clearer; object allocates a certain chunk of memory and it is also responsible of freeing it. C has *structs*, but they do not have common routines like C++ objects, such as constructors and destructors.

The mindset between the two languages is completely different, although both C and C++ could be programmed in a very similar manner – if needed [6]. C is a subset of C++ features, but as the years have gone by, they have become a completely different languages; from minor differences to major differences. The newer standards differentiate the languages even further [6].

7.3 Improving existing code

This sub chapter estimates the benefits and drawbacks of using C++ in a modern embedded development.

7.3.1 Binary literals

The new C++14 standard offers binary literals, which might be useful for embedded development. They are calculated compile time. C has no official support for binary literals. Only compiler dependent extensions are available.

7.3.2 Smart pointers

When using `malloc/free` in C, or `new/delete` in C++ – it introduces a problem: who is responsible for the allocated memory? Is it caller's task of freeing the allocated memory? In old C programs, this is usually implicitly stated in function names, such as `create_func` or `new_func`, which return a pointer to an allocated memory.

The new C++11 standard introduces smart pointers. They completely eliminate the use cases of naked `new`. Newly written C++11 code should never use naked `new` operator for dynamically allocating the memory. Only new operator inside smart pointers is recommended. [6]

The C++11 standard introduced three important smart pointer types, which might be useful for embedded programmer: `std::shared_ptr`, `std::weak_ptr` and `std::unique_ptr`. The `std::unique_ptr` has zero overhead and frees the resource when it exits scope [6]. It can be explicitly moved from scope to scope with `std::move`. It is also a new C++11 feature.

The `std::shared_ptr` is meant to be used when the resource is being accessed from multiple scopes. `The std::shared_ptr` has reference count, which means every time the pointer is copied, the reference count is increased, and every time shared pointer instance exits scope, the reference count is decreased. When the last reference of `std::shared_ptr` exits the scope, the memory under `std::shared_ptr` is being freed.

The `std::weak_ptr` is designed to be used with `std::shared_ptr` for avoiding circular references. Circular references could happen (Figure 5), if the resource is indirectly referenced back to itself, which causes memory leaks.

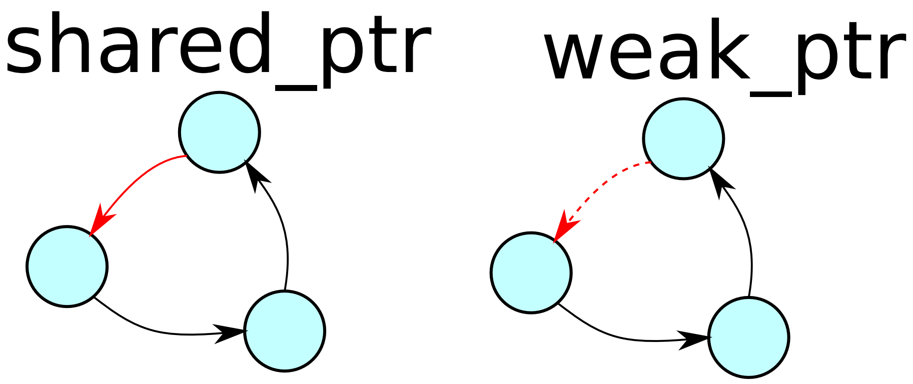


Figure 5. Weak pointer breaks the circular reference.

This is because the reference count will never achieve zero, which means the memory is never freed. The `std::weak_ptr` was designed for breaking the circular references.

The `std::shared_ptr` introduces a small overhead [6], due to its reference counting characteristics. Pointers and references should still be used in modern C++. If the resource owning semantics is explicitly wanted, then the smart pointers should be used.

7.3.3 Ring buffer

As the ring buffer chapter mentioned; C standard library is problematic. It only supports string manipulation functions via character arrays, where the memory is always contiguous. For e.g.: it is not possible to feed C standard library functions with strings made of linked lists. It always must be a C array.

In C++, it is possible – without re-implementing the whole library. The only requirement for a class is to implement `begin` and `end` iterators [6], so that the STL algorithms can iterate any data type container.

The C++14 standard has a singly linked list implementation known as `std::forward_list` [6], and it has already implemented `begin` and `end` iterators. All the STL data types have a default iterator implementation. The `std::forward_list` dynamically allocates memory, so it might not be suitable for embedded applications.

7.4 Time critical software

Time critical software is usually categorized under two types [7]:

- hard – missing a deadline is a total system failure.
- soft – missing a deadline reduces the QoS, but is still usable.

Many C++ features use dynamic allocations, and their inclusion to projects should be carefully considered. Containers, such as `std::array<T>`, uses heap allocation (Figure 6) for the element and node data.

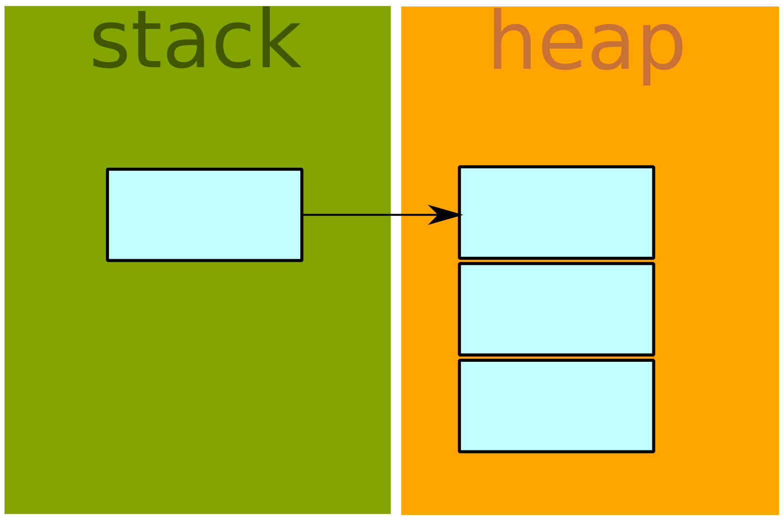


Figure 6. Memory layout for a `std::array<int> arr{1,2,3}`.

It is likely for an embedded developer to end up using only subset of features in C++ [7]. The Standard Template Library (STL) uses heap allocation on most of its default containers, which limits the use cases of STL in real time systems.

In STL, there is one exception for allocating memory for containers: `std::array<T,N>` (Figure 7). It allocates a fixed memory portion from the stack.

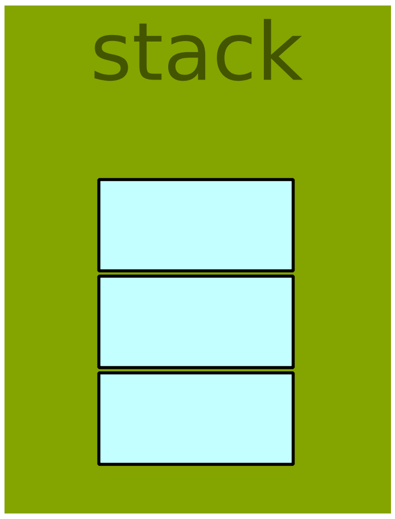


Figure 7. A container `std::array<int,3> arr{1,2,3}`.

As mentioned earlier: STL uses heap allocation for most of its default containers. Fortunately, overriding the default allocators in C++ is possible, by doing an own implementation of the allocator class. Implementing any allocator for any container in

STL is possible. For example: memory pool allocation scheme for a queue could have been used.

7.4.1 Virtual functions

The virtual functions should be avoided when dealing with real time systems, because the virtual function location needs to be discovered at run-time from the vtbls virtual table. The vtbl table array consists of function pointers. Their implementation is completely compiler specific [7]. Virtual functions are a low cost operation, so they might be useful in some cases.

7.4.2 Function pointers

The C++11 introduced a `std::function`, which represents a modern function pointer. It has a small overhead compared to naked C function pointers, but they are easier for compilers to inline [7]. The C function pointers are stateless, unlike `std::function`.

The function and its parameters state could be saved within a `std::bind`. The state machine function pointer implementation could have been improved; implementing it as a `std::function` table, and binding each function to it. [7]

The fan controller struct implemented all of the fan actions as a separate function, indexed from 0 to 7. With `std::function` and `std::bind`, only one function with different parameter bindings could have been achieved.

All of the functions in fan controller were short, so they could have been implemented as lambdas – a new C++11 feature. The `auto` keyword could have been used for figuring out the types automatically by the compiler. It has zero overhead and is also a new C++11 feature [7].

7.4.3 RTTI

In hard real time applications using run-time type information (RTTI) is forbidden, since the operations are done run-time – as the name states [7].

RTTI could be disabled by the compiler flags [7]. Disabling it removes the possibility of using `std::dynamic_cast`, which downcasts a derived class if it is possible. If the base class has virtual methods, `std::dynamic_cast` could be used for figuring out which virtual functions were implemented within the derived class.

7.4.4 Exceptions

In hard real time applications, usage of exceptions should be avoided, since their execution path will be discovered during run-time [7].

8 CONCLUSION

The goal of the thesis was to implement a fan controller controlled via a USB-CDC. It was first planned to achieve a physical working prototype, but only the software part was achieved during the thesis. The Arduino Leonardo hardware was used as a base prototype for development, but Arduino's software stack was not used during the thesis. AVR Libc and LUFA were used as a software stack. AVR Libc has not implemented STL, which means some parts in the thesis are not possible to implement in AVR.

The thesis describes the design process of a fan controller and its design considerations, particularly in the AVR platform. The general *pro et contra* for implementing data structures in embedded systems are considered. The existing software, alternative and possibly better implementation were weighted. The thesis describes implementing a lexical parser and queue in depth.

Lastly, the existing C program is reflected to a modern C++ development, and how the existing code could have been improved in C++. C and C++ development differences are compared in general, in respect of embedded systems. Safety, performance and memory considerations are discussed in the C++ chapter. The C++ and hardware chapters were purely theoretical.

REFERENCES

[1] Free Software Foundation. AVR Libc. Consulted 10.5.2017 http://www.nongnu.org/avr-libc/user-manual/

[2] Axelson, J. 2009. USB Complete: The Developer’s Guide. Fourth edition. Madison: Lakeview Research LLC.

[3] Gunther, J. C. 2014. Algorithm 938: Compressing circular buffers. ACM Transactions on Mathematical Software (TOMS) Vol. 40 (2) No. 17/2014, 1-12.

[4] Microchip. 2015. ATmega16U4/ATmega32U4. Datasheet. Consulted 10.5.2017 http://www.atmel.com/Images/doc7618.pdf

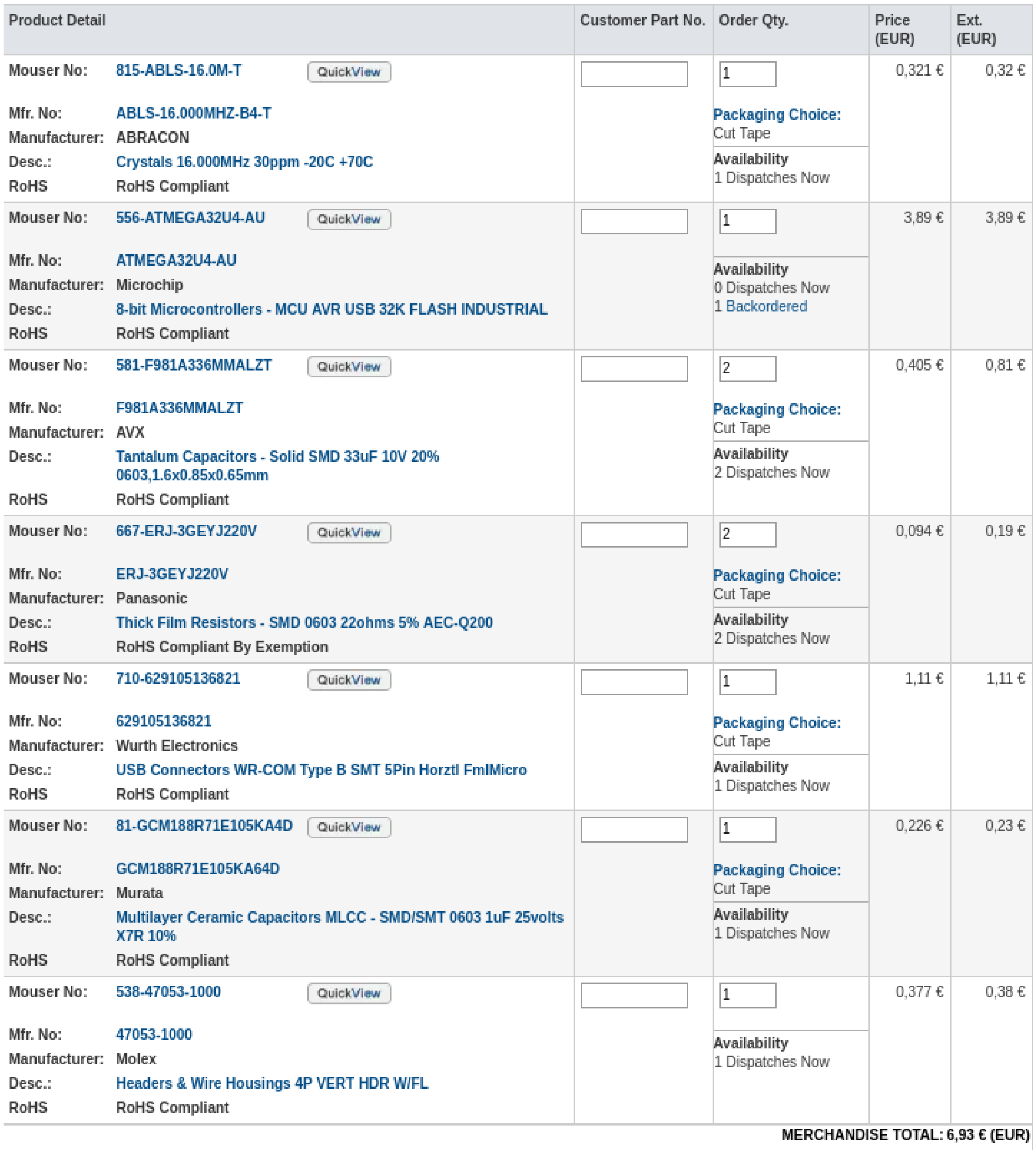
[5] Stourstrup, B. 2014. Are Lists evil? Consulted 10.5.2017 https://isocpp.org/blog/2014/06/stroustrup-lists

[6] Stroustrup, B. 2013. The C++ Programming Language. Fourth edition. Boston: Addison-Wesley.

[7] Stroustrup, B. 2013. A Tour of C++. Fourth edition. Boston: Addison-Wesley.

[8] Meyers, S. 2012. Effective C++ in an Embedded Environment. Second edition. Presentation material.

Part list



Product Detail		Customer Part No.	Order Qty.	Price (EUR)	Ext. (EUR)
Mouser No:	815-ABLS-16.0M-T QuickView		1	0,321 €	0,32 €
Mfr. No:	ABLS-16.000MHZ-B4-T		Packaging Choice: Cut Tape		
Manufacturer:	ABRACON		Availability 1 Dispatches Now		
Desc.:	Crystals 16.000MHz 30ppm -20C +70C				
RoHS	RoHS Compliant				
Mouser No:	556-ATMEGA32U4-AU QuickView		1	3,89 €	3,89 €
Mfr. No:	ATMEGA32U4-AU				
Manufacturer:	Microchip		Availability 0 Dispatches Now 1 Backordered		
Desc.:	8-bit Microcontrollers - MCU AVR USB 32K FLASH INDUSTRIAL				
RoHS	RoHS Compliant				
Mouser No:	581-F981A336MMALZT QuickView		2	0,405 €	0,81 €
Mfr. No:	F981A336MMALZT		Packaging Choice: Cut Tape		
Manufacturer:	AVX		Availability 2 Dispatches Now		
Desc.:	Tantalum Capacitors - Solid SMD 33uF 10V 20% 0603,1.6x0.85x0.65mm				
RoHS	RoHS Compliant				
Mouser No:	667-ERJ-3GEYJ220V QuickView		2	0,094 €	0,19 €
Mfr. No:	ERJ-3GEYJ220V		Packaging Choice: Cut Tape		
Manufacturer:	Panasonic		Availability 2 Dispatches Now		
Desc.:	Thick Film Resistors - SMD 0603 22ohms 5% AEC-Q200				
RoHS	RoHS Compliant By Exemption				
Mouser No:	710-629105136821 QuickView		1	1,11 €	1,11 €
Mfr. No:	629105136821		Packaging Choice: Cut Tape		
Manufacturer:	Wurth Electronics		Availability 1 Dispatches Now		
Desc.:	USB Connectors WR-COM Type B SMT 5Pin Horztl FmlMicro				
RoHS	RoHS Compliant				
Mouser No:	81-GCM188R71E105KA4D QuickView		1	0,226 €	0,23 €
Mfr. No:	GCM188R71E105KA64D		Packaging Choice: Cut Tape		
Manufacturer:	Murata		Availability 1 Dispatches Now		
Desc.:	Multilayer Ceramic Capacitors MLCC - SMD/SMT 0603 1uF 25volts X7R 10%				
RoHS	RoHS Compliant				
Mouser No:	538-47053-1000 QuickView		1	0,377 €	0,38 €
Mfr. No:	47053-1000				
Manufacturer:	Molex		Availability 1 Dispatches Now		
Desc.:	Headers & Wire Housings 4P VERT HDR W/FL				
RoHS	RoHS Compliant				

MERCHANDISE TOTAL: 6,93 € (EUR)