TALLINN UNIVERSITY OF TECHNOLOGY

School of Information Technologies

Rain Sarapuu 192510IVEM

Performance Improvement of Electronic Control Unit for Formula Student Electric Car

Master's thesis

Supervisor: Mairo Leier

PhD

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Rain Sarapuu 192510IVEM

Elektrilise tudengivormeli elektroonilise juhtüksuse täiustamine

Magistritöö

Juhendaja: Mairo Leier
Doktorikraad

Tallinn 2021

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Rain Sarapuu

10.05.2021

Abstract

Aim for this thesis work is to develop an improved version of Electronic Control Unit based on the ECU used in student formula car FEST20. ECU’s main task is to control formula car’s motor inverters. Purpose will be to implement a real-time operating system to provide prioritised pre-emptive scheduling to achieve more time-critical system. Due to testing purposes, one of the most important functions running on ECU is optimised to get performance gain in task execution times. Previous ECU solution is going to be analysed. Analysis is necessary to gain an overview on previous system’s performance. New improved solution is also analysed, therefore this work provides a comparison between both implementations to indicate main differences achieved. As a result of this thesis a refined ECU is provided which uses real-time operating system and is able to compute control system setpoints with higher frequency.

This thesis is written in English and is 39 pages long, including 7 chapters, 25 figures and 4 tables.

Annotatsioon

Elektrilise tudengivormeli elektroonilise juhtüksuse täiustamine

Antud lõputöö eesmärgiks on arendada parendatud versioon elektroonilisest juhtüksusest (ingl k – *Electronic Control Unit, ECU*) tudengivormelis FEST20 kasutusel oleva ECU põhjal. ECU põhiliseks ülesandeks on vormeli mootorite inverterite juhtimine. Eesmärgiks on reaalaja operatsioonisüsteemi kasutusele võtmine, et määrata programmikoodi funktsioonidele prioriteedid. Vastavalt prioriteetidele ajastab reaalaja operatsioonisüsteem funktsioonide täitmist, arvestades seejuures funktsiooni jooksutamise eesõigust määratud prioriteetidest tulenevalt, et parandada süsteemi ajakriitilisusest kinni pidamist. ECU kõige tähtsamat funktsiooni optimeeritakse, et lühendada selle jooksutamiseks kuluvat aega, mille tulemusena saab testimise eesmärgil tõsta selle funktsiooni arvutussagedust. Eelmise ECU jõudlusest parema ettekujutuse saamiseks seda analüüsitakse. Samuti analüüsitakse uut täiustatud versiooni ning võrreldakse seda vanaga, et leida põhilised saavutatud erinevused nende vahel. Käesoleva lõputöö tulemusena valmib reaalaja operatsioonisüsteemil põhinev ECU, mis võimaldab juhtimissüsteemi funktsiooni arvutussageduse tõstmist.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 39 leheküljel, 7 peatükki, 25 joonist, 4 tabelit.

List of abbreviations and terms

AMS	Accumulator Management System
API	Application Programming Interface
BCET	Best-Case Execution Time
CAN	Controller Area Network
CPU	Central Processing Unit
DP-FPU	Double Precision Floating Point Unit
DRS	Drag Reduction System
ECU	Electronic Control Unit
Flash memory	Non-volatile computer memory storage medium
FSTT	Formula Student Team Tallinn
GCC	GNU Compiler Collection
GSS	Ground Speed Sensor
HAL	Hardware Abstraction Layer
IDE	Integrated Development Environment
IMU	Inertial Measurement Unit
MCU	Microcontroller
PCB	Printed Circuit Board
RTOS	Real-Time Operating System
SRAM	Static Random Access Memory
SP-FPU	Single Precision Floating Point Unit
TS	Tractive System
VAC	Voltage of Alternating Current
VDC	Voltage of Direct Current
WCET	Worst-Case Execution Time

Table of contents

List of figures

List of tables

1 Introduction

This thesis work has been done based on a project from Formula Student Team Tallinn (FSTT). Formula Student series consists of product development and engineering competitions held worldwide in which teams compete with a small-scale one-seated formula racing car. Cars must be proven in dynamic racing events and design defending event. FSTT consists of students from Tallinn University of Technology and Tallinn University of Applied Sciences. The team has been active since 2006. During first 7 years the team focused on developing combustion engine formula cars, but since 2013 moved to using electric motors. Every season there is a new electric formula car being designed, built, and tested to compete in mostly European competitions during summer period.

This project of building student formula car helps university students to improve their knowledge, skill, know-how and practice what they have learned from school. It gives students valuable experience from engineering perspective as most of the vehicle is self-developed. Additionally, there is a chance to improve one's ability to justify the design decisions of certain systems by defending their design to experts of the field. To date FSTT has been successful and achieved high ranking places, despite competition being very high. To maintain and further improve results, team must develop, analyse, and refine systems and parts of which formula cars consist each year.

Current thesis is focusing on providing improved version of one of those sub-systems called the Electronic Control Unit (ECU), which is an embedded system. It is mainly responsible of communicating with motor controller to control motor inverters. It also performs safety checks and commands vehicle to stop in case of error. Program functions in current solution's source code are executed using self-developed timer-based system which is a non-time-critical and non-prioritised timing system. To make the system more reliable and efficient, a real-time operating system is essential which is going to be implemented as a result of this thesis. The goal is to provide pre-emptive time-critical task execution with strict deadlines, prioritised tasks, and deterministic scheduler to enhance ECU's functional safety and efficiency.

One important component of ECU's software is control system. This is used in the process of controlling motors to determine how formula car performs on track. It is responsible of calculating setpoints containing information essential for motor control. Setpoints are computed and transmitted to vehicle's motor controller by ECU with specific frequency. The frequency of calculation and transmission could be raised to acquire test data which can be used to analyse and further determine optimal refresh rate of motor control. That could improve vehicle's dynamic driving performance. Together with implementing RTOS (Real-Time Operating System) the goal is to develop ECU with improved performance to be able to handle higher control system computation frequencies.

This thesis includes analysis of previous ECU solution developed for FEST20 formula car and provides possible new solutions based on that. It focuses on implementing, testing RTOS and discusses outcomes. New system with refined performance is analysed in comparison with old system to highlight gained benefits. Tests and analysis have been carried out in similar conditions:

- FEST20 formula car used.
- Control system developed for FEST20 used [1].
- FEST20 ECU PCB and FEST21 ECU PCB used (differences on these PCBs do not affect results achieved).

2 Description of electric formula car

The formula car is a four-wheel drive vehicle depicted on Figure 1, which has a motor in every wheel. Motors are controlled by inverters inside the controller. The controller also has a logic board, which is responsible for receiving input setpoints, controlling motor inverters, getting feedback information from motor encoders, and transmitting current status and motors information as feedback to the electronic control unit. Due to series rules, the maximum output power from accumulator container which could be used for motors is 80 kW [2, p. 72]. To keep power usage under that level, a power limiter function has been built into control system.

Figure 1. Electric formula car FEST20 [3].

Racing tracks at competitions are usually made up of tight turns, slalom elements and there are not so many long straight sections because proving cornering ability of the race cars has the most importance. Nevertheless, maximum speeds reached by drivers can go up to 120 km/h on some tracks. In addition, considering formula car’s sharp acceleration and deceleration capabilities, safety is a huge concern when designing parts of the formula car.

The formula cars are powered by a battery pack, which provides power for all electric systems in the vehicle. The vehicle's electrical system is divided into LVS (Low Voltage System) and TS (Tractive System). As defined in series rules [2, p. 54] all electric systems with occurring maximum voltages under 60 VDC or 25 VAC RMS (root mean square) are considered as low voltage systems and anything above 60 VDC is part of TS. Main supply voltage used in low voltage system is 24 VDC. This is the operating voltage for most of the systems in LVS including ECU.

All digital systems in LVS communicate with each other through vehicle's main CAN (Controller Area Network) bus. Thus, main CAN bus has the following uses:

- Transfer data required by the control system to ECU
- Transfer data for dashboard to display system states and parameters
- Transfer data to log most of the messages
- Transfer data to indicate of error states

CAN communication is used widely in automotive industry because it is a robust protocol that provides possibility for many devices to be connected to the same bus. Bus speed and cost add up as a benefit when choosing a means of communication. Additional advantage is the prioritised messages, which ensure good timing and most important information to be transferred first as stated in this article [4, p. 721]. The formula car's digital systems communication is visualised on Figure 2.

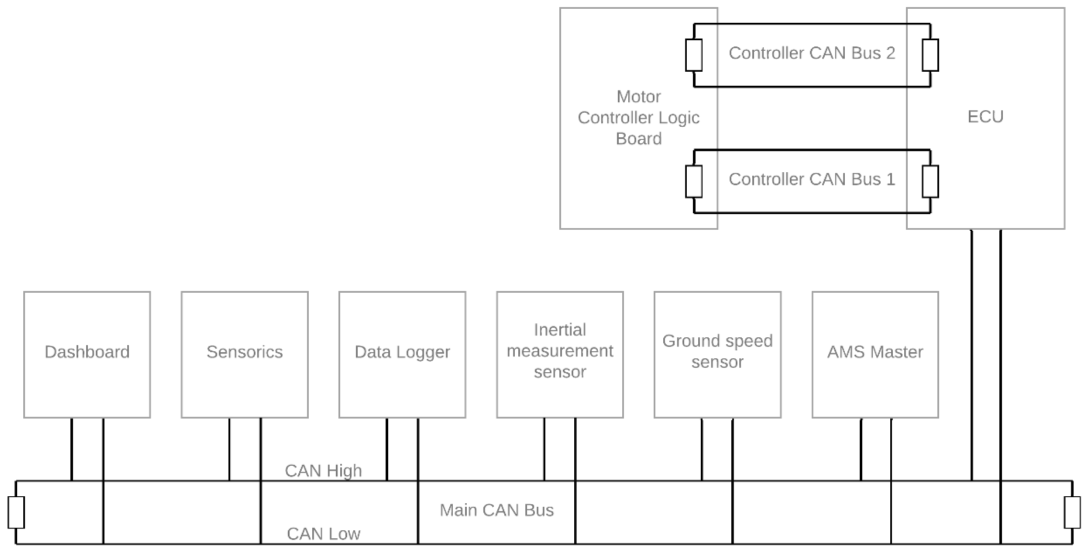


Figure 2. Overview of CAN buses to which ECU is connected.

Following section further describes digital electronics systems in low voltage system which are connected to ECU.

2.1 Sub-systems interconnected to ECU

Dashboard is a self-developed module consisting of a display and PCB, which main responsibility is to provide a user interface for the driver and assisting engineers. Display indicates general and most important information about the status of the vehicle and sub-systems. User interface offers possibility to configure input parameters of control system and to control other systems. Dashboard communicates with other PCBs using CAN. ECU reads dashboard input messages which are required for control system.

Accumulator management system (AMS) is used to monitor the state of accumulator package. The system consists of self-developed master PCB and slave PCBs. There is one slave PCB for every accumulator module and these PCBs are responsible for acquiring important information such as cell voltages and temperature. AMS master PCB collects the information from slave PCBs and determines the state of accumulator. In case of an error high voltage tractive system is deactivated. AMS master PCB communicates with other systems on the main CAN bus. ECU uses important message content from AMS for error checks and as control system input.

Sensorics PCB is a self-developed system gathering data from external sensors such as pedals' positions, steering wheel angle, brake pressure, temperatures. It conditions signals, processes digitised values and transmits data onto the main CAN bus. All the information coming from sensorics is important to be able to drive the formula car, thus ECU receives those messages and passes them for control system.

Ground speed sensor (GSS) is a self-developed sensor system. The main goal is to measure vehicle's longitudinal and lateral velocity relative to the ground. This information is processed in ground speed sensor and sent to the main CAN bus. The data is used by ECU in vehicle's control system.

Inertial measurement unit (IMU) is responsible for providing data about formula car's orientation like acceleration, deceleration, and gyroscopic information. This system transmits data onto the main CAN bus for logging and ECU to use as control system input.

Data logger is used on the main CAN bus. Depending on the configuration it acquires all necessary messages content which could be later viewed and analysed from data logging software. Logging data is a very important feature as engineers get most of the feedback from the data and graphs logs contain. Information in logs is used to verify if a system operates as expected or identify the cause of failures and system error states.

2.2 Description of Electronic Control Unit

Electronic control unit is a self-developed PCB, which consists of hardware, peripherals, functionality-based application code, and control system. ECU serves a main purpose of controlling motor inverters. The controlling is done through CAN communication between ECU and controller. ECU is responsible for transmitting setpoints to controller containing information about status, motor speeds and torques [5]. Controller sends feedback which is in turns used when calculating new setpoints by control system. Control system needs many different parameters from different systems across the formula vehicle.

ECU is additionally responsible for the following:

- Acquire temperatures of rear motors, cooling radiators and rear brake supports.
- Control water pump
- Control cooling fans
- Control buzzer used for ready-to-drive sound [2, p. 80]
- Control brake light
- Control DRS (Drag reduction system)
- Switch power for motor controller’s logic board and cooling system’s water pump

Considering CAN bus loads and motors and controller manufacturer’s recommendations, ECU communicates with controller using two separate CAN buses as seen on Figure 2. Therefore, each bus has the setpoint and feedback information of two motors.

All the processing on ECU is done by an MCU (microcontroller). It belongs to the STM32F7 series [6] of microcontrollers which are produced by STMicroelectronics. It is a 32-bit MCU and has a maximum CPU speed of 216 MHz. It hosts an ARM Cortex M7 core.

2.3 Description of control system

Control system is a model-based program application, which is made using MATLAB Simulink [7] environment. Simulink has the advantage to build the program as a model using different blocks with various functionality. This means that creation of complex algorithms and programs is faster and easier than writing code. Thus, engineers can focus more on developing functionality of the program. Additional benefit provided by Simulink is the ability to simulate designed model. This gives feedback about how system is operating. With Simulink Embedded Coder [8], it is possible to automatically generate C code from user model. Code generation has many configuration options and generated code can be directly used on an embedded processor.

A simplified block diagram of the control system used throughout current thesis is depicted on the following Figure 3 to give an overview. The control system used in current thesis was developed for Formula Student Team Tallinn's formula car FEST20 and has not been made by the author of this thesis [1].

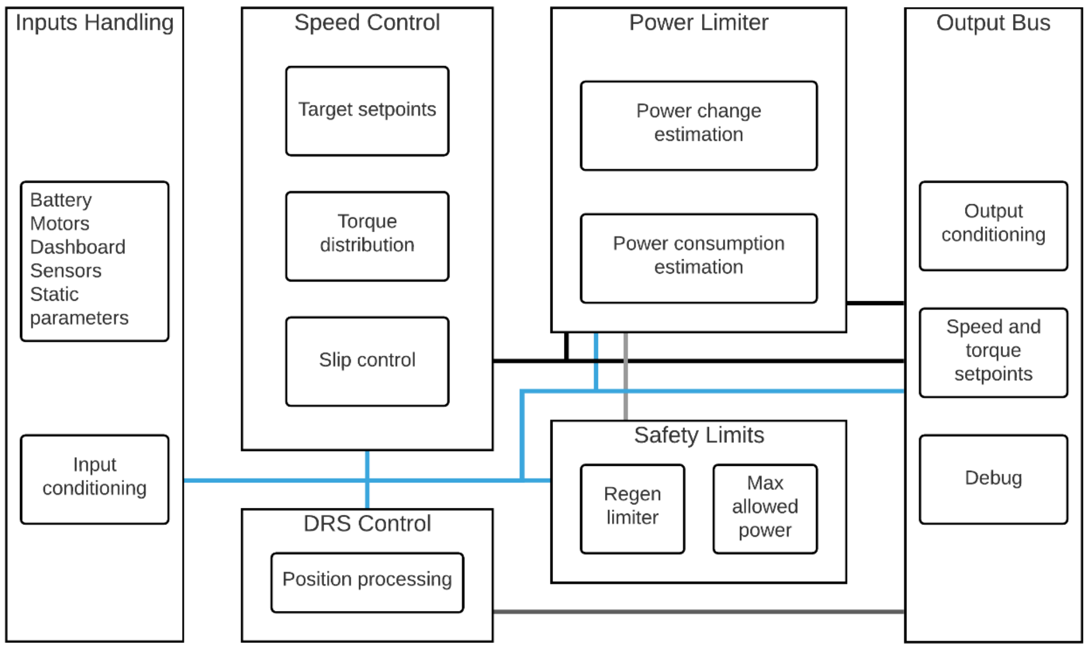


Figure 3. Simplified block diagram of control system.

Inputs handling section has a data bus for all input data signals. ECU is responsible for updating input data with every iteration. The data comes from AMS, dashboard, sensorics PCB, IMU, GSS, static pre-defined parameters and motors' feedback. Conditioning of input data is done in this section [1].

Speed control section calculates target speed and torque limit setpoints based on pedal position and steering wheel angle. It distributes torques for each motor depending on acceleration or deceleration forces. Slip control block adjusts setpoint values according to wheel slip [1].

Safety limits block calculates power limits for regenerative braking and maximum output power limit allowed based on accumulator inputs [1].

Power limiter section is responsible for estimating change in output power every iteration. It calculates estimated power consumption and must make sure that formula car does not use more than 80 kW form the accumulator to comply with rules [1] [2, p. 72].

Output bus section is used to condition output data which are speed and torque limit setpoints for every motor. ECU transmits these setpoint values onto two CAN buses between controller and ECU [1].

DRS control block calculates desired position state of vehicle’s rear upper wing profile [1].

3 Analysis of previous solution

To understand current situation with function execution times, determinism, the solution of FEST20 electronic control unit had to be analysed. As this ECU is not based on RTOS the test methods had to be selected such that same characteristics could be found later with the analysis of new solution. Before addressing testing and analysis results, next sections are covering target goals of analysis and possible testing techniques.

3.1 Aim of analysis

In different domains and fields, the time-criticality of completing software functions in certain timeframe is of varying importance as illustrated on Figure 4. For example, in aircraft control and avionics the task execution times for control software are strictly defined and the deadlines must be met. In this case vehicle control also has mostly hard time requirements because the operation of vehicles is safety-critical [9, p. 2].

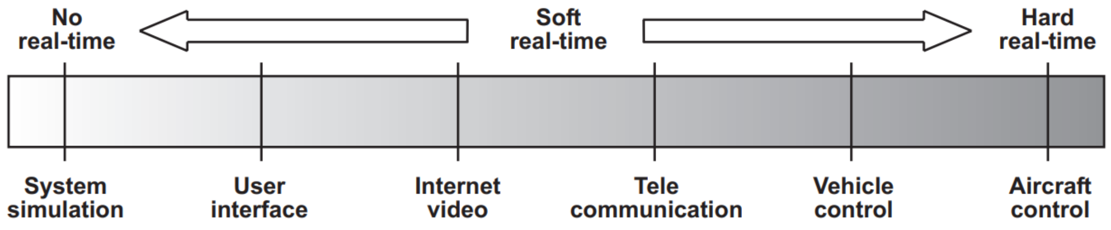


Figure 4. Timing criticality in different fields [9, p. 2].

3.2 Methods used

This section describes methods used for analysing previous ECU’s performance. These methods are going to be used for analysing new solution as well.

3.2.1 WCET analysis

Worst-case execution time (WCET) analysis is one of the main ways of analysing a real-time operating system. This analysis provides insight of how much time each function takes to execute. This means the start and stop times of each function are measured to get the elapsed time. Worst-case execution time of a certain program section is the longest recorded time to execute that section. WCET analysis typically also provides BCET (best-case execution time) value and generally gives an overview of the average execution times and the distribution of execution occurrences [9, pp. 1-4]. A good example of WCET analysis is provided on Figure 5.

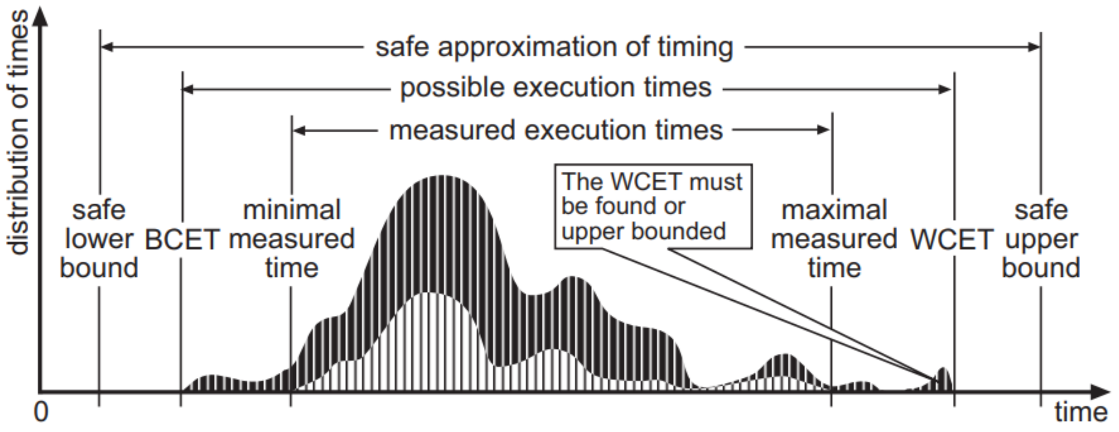


Figure 5. WCET analysis - distribution of execution times [9, p. 4].

3.2.2 Scheduling jitter analysis

Scheduling jitter is the deviation of task's actual starting time from ideal (nominal) starting time [10, p. 850]. There are different methods used for jitter analysis. Periodic task's jitter can be obtained by acquiring the starting time of the task, knowing the ideal desired execution period, and calculating deviations between when the task should start and when it really starts. Jitter specification for a function is usually given as worst-case jitter between the minimum and maximum deviations from the nominal execution period [11, p. 11].

Most of the functions in ECU's source code are periodic, which means scheduling jitter analysis is a suitable option to characterise the system. This is because scheduling jitter provides an overview of how consistent is the task execution in terms of determinism.

3.3 Possible execution time measurement techniques

Program code execution time measurement can be done using many different techniques, tools, and equipment. Some of the common ways presented in this article [9, pp. 14-15] are the following:

- **Oscilloscope and logic analysers** – these methods are beneficial in a way that they do not alter execution times of the program while measuring running time.

Oscilloscope can be used to measure the state of an output pin or LED, which is being toggled in the program section of interest. Execution time can be measured from the state changes. Logic analysers on the other hand monitor system's data bus for specific instructions and calculate execution time based on that [9, p. 14].

- **High-resolution timers** – this method uses timers which are part of the system. It requires modifying program code in order to capture function starting and stopping times. Thus, this method uses processors resources and therefore may slightly affect execution times [9, p. 14].
- **Hardware traces** – this method uses built-in hardware tracing and debugging features of the system for measuring program code executions. Widely-used interface for that is the JTAG (Joint Test Action Group) [9, p. 14].
- **Profilers** – this method can be used if it is supported by the compiler. Execution time is measured based on hardware timers. Measurement results can be imprecise depending on the working principle [9, p. 15].
- **Operating system facilities** – this method can be used in the absence of an operating system which has to provide features for timing measurement of specified functions. Usually this method needs a hardware timer for measuring [9, p. 15].
- **Simulators** – this method simulates the processor for timing measurements. Developing and implementing a simulator to get correct results is very difficult [9, p. 15].

3.4 Chosen measurement techniques

Considering available tools and possibilities the most realistic choice was to use high resolution timers, which are part of MCU used on ECU. Additionally, this solution can be integrated into the system in most suitable way because the timer counter values can be easily logged with vehicle's data logger and later exported into Excel for further analysis. Timer *TIM2* has high resolution of 32 bits. It had to be configured to be able to measure execution times with 1 μs (microsecond) resolution.

On Figure 6 the configuration with prescaler and period values is shown. *TIM2* peripheral timer clock is configured to 108 MHz. Prescaler value for this timer is set to 107 (timer clock is actually prescaled by 107 + 1 in this case) to get a timer period of 1 μs. The period variable in timer configuration is set to 1999999999 which equals 2000 seconds. This means the timer counter is counting for 2000 seconds and makes an incrementation after every 1 µs. If 2000 seconds is reached the counter resets its value to 0 and starts incrementing again. 2000 second period is long enough for even track testing. 1 µs resolution is chosen based on current task execution frequencies to get meaningful data and the fact that jitter analysis results are expected to be in the microsecond range.

```
TIM_HandleTypeDef htim2;
htim2.Instance = TIM2;
htim2.Init.Prescaler = 107;
htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
htim2.Init.Period = 1999999999;
htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
if (HAL_TIM_PWM_Init(&htim2) != HAL_OK)
{
       Error_Handler();
}
```

Figure 6. Configuring microcontroller’s high resolution timer peripheral.

3.4.1 Acquiring test data

The data was measured by capturing timer counter values at desired program locations in program code. To get current counter values HAL (Hardware Abstraction Layer) driver’s timer function was used, which returned counter’s current value. Functions to capture the current value were created and called before and after execution of the task under measurement. These functions are shown on Figure 7.

```
void set_bm1_start() {
       BM1_start = __HAL_TIM_GET_COUNTER(&htim2);
}
void set_bm1_end() {
       BM1_end = __HAL_TIM_GET_COUNTER(&htim2);
}
int get_bm1_start() {
       return BM1_start;
}
int get_bm1_end() {
       return BM1_end;
}
```

Figure 7. Functions for capturing timer counter current value.

Data was gathered with the help of vehicle’s data logger. Captured counter values were sent to formula car’s main CAN bus for logging. Data logger’s configuration was modified to accommodate new variables. The raw testing data presenting timer counter values as seen in data logger's computer software is depicted on Figure 8. After testing session suitable timeframe of necessary data and timestamps was exported as excel document. Further analysis was done in excel.

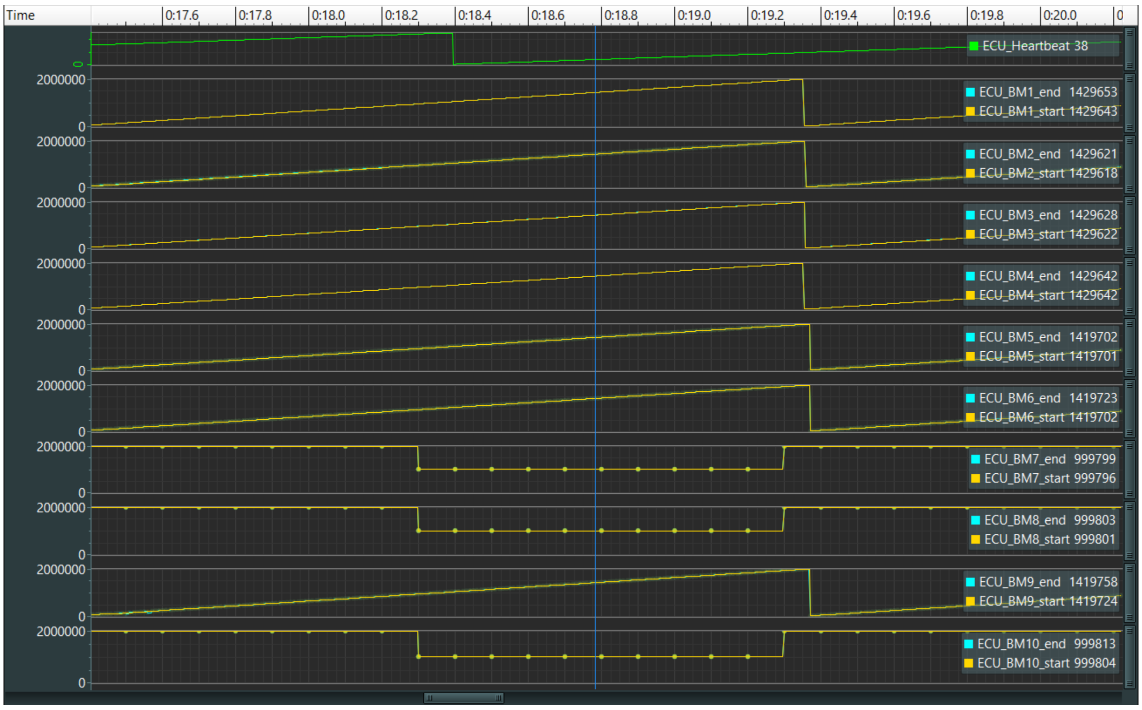


Figure 8. Raw timer counter data in data logger software.

To complement high resolution timer measurement, some of the measured values were additionally validated by measuring with an oscilloscope HAMEG HMO 3524 [12]. However, this could not be done in case of track driving.

3.5 Analysis results

Previous ECU's program code has main functions which are called by the timing system in program's *int main(void)* function. These functions all serve their own purpose, whether it be controlling systems connected to ECU, calculating control system setpoints or reading and processing ADC (Analog Digital Converter) channels' input signal values. Primary target of testing was to determine time required for running each program code function (further referred to as a task). Based on running times task execution analysis was made together with WCET and jitter analysis for control system.

3.5.1 Task execution time

After analysing data in excel it was clear that control system task takes the longest time to run which can be seen on Figure 9. For this reason, other functions were added together and depicted as one bar. For every task minimum, average, median and maximum execution times were found. In case of maximum column, longest execution times of every task were added on top of each other regardless of the timestamp they were taken because the intention is to determine the worst possible situation.

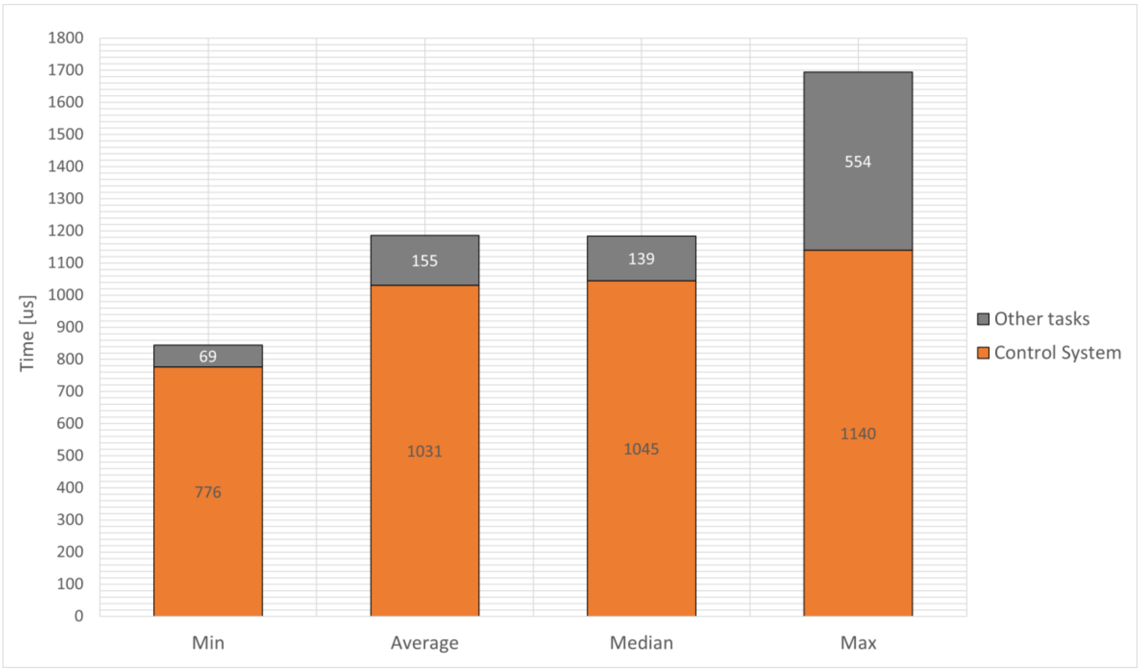


Figure 9. Task execution times on the previous ECU.

Derived from the maximum column on Figure 9, in worst case the whole program iteration could take 1694 µs, which means iteration computation frequency of approximately 590 Hz. This could be safe when using previous ECU's control system execution frequency of 200 Hz, but it is quite close to potential 500 Hz execution frequency of new improved ECU's expected performance, without much room for reserve. Considering remarkable fluctuations in control system execution times and possible increased performance need caused by further development, providing higher safe upper bound estimation is essential [9, p. 4].

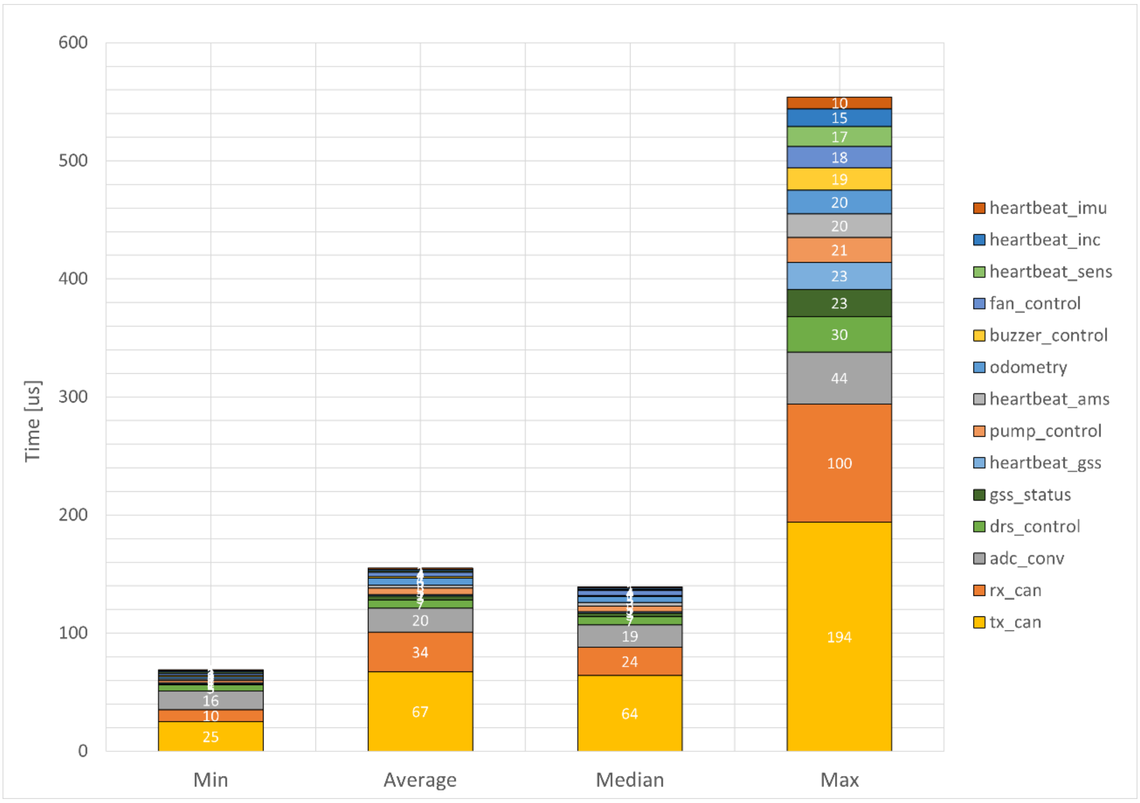


Figure 10. Task execution times without control system on the previous ECU.

Execution times of other tasks excluding control system can be seen on Figure 10. It seems *tx_can* and *rx_can* take the longest to run when compared to other tasks. In maximum possible conditions other tasks can play a significant role in execution time of whole iteration, but in minimum, average, and median cases other functions do not affect program iteration that much.

3.5.2 WCET analysis of control system

To give a good overview of WCET and BCET the graph seen on Figure 11 was created. This graph shows the distribution of execution times for control system. The horizontal axis is for task execution time in microseconds and vertical axis shows how many times a certain execution time has occurred. This graph also gives an understanding of average execution times of certain task. In this case average execution times of control system are from 1040 - 1100 µs. The BCET is 776 µs and WCET 1140 µs. It seems that in some conditions control system execution times are also lower in the range of 810 – 890 µs. This could be due to control system not using certain sensor data in its calculations during those iterations.

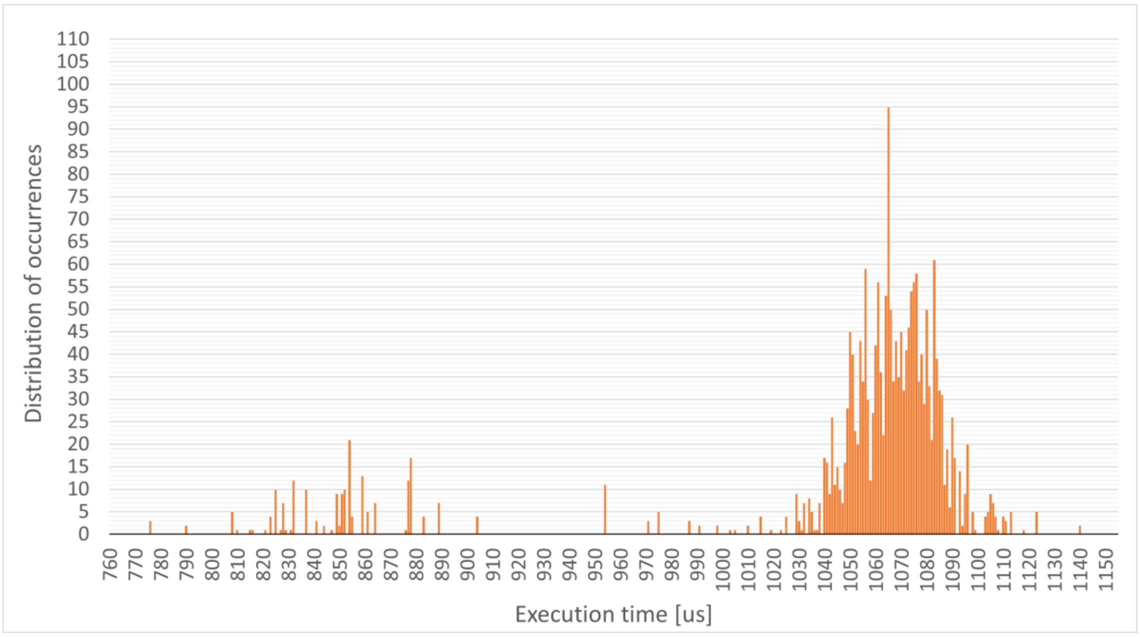


Figure 11. Distribution of control system execution times.

3.5.3 Scheduling jitter analysis of control system

In previous FEST20 ECU control system was a periodic task with execution frequency of 200 Hz which means that in ideal case the task should start processing after every 5000 µs. This would mean there is no scheduling jitter, but in practice there are deviations in the execution starting times. Plot on Figure 12 depicts scheduling jitter analysis of 1000 task instances. The green line indicates the ideal instance starting time from last instance execution. Red dots demonstrate current task instance's starting time from last execution. The jitter of one instance is the difference between current instance's starting time and ideal (nominal) starting time i.e., if current instance's starting time is 5025 µs, it means that 25 µs is the jitter in that case because 5000 µs is the ideal start time.

For a hard real-time operating system, the jitter should typically be from some microseconds to a few tens on microseconds as stated in this article [11, p. 10]. From the graph on Figure 12 it can be seen that many task instances have jitter above 30 µs and some abnormal instances even have in the range of 125 – 185 µs. Those can arise in a situation where ECU receives CAN bus messages due to interrupts and therefore executes message processing functions before handling control system task. Maximum deviation over the nominal period time is 185 µs and under nominal -3 µs. This means the worst-case jitter for control system task execution is 188 µs which is quite significant and can affect system performance. This is the result of using sequential non-prioritised scheduling, implementing RTOS could be a solution for this problem and should decrease the unwanted jitter.

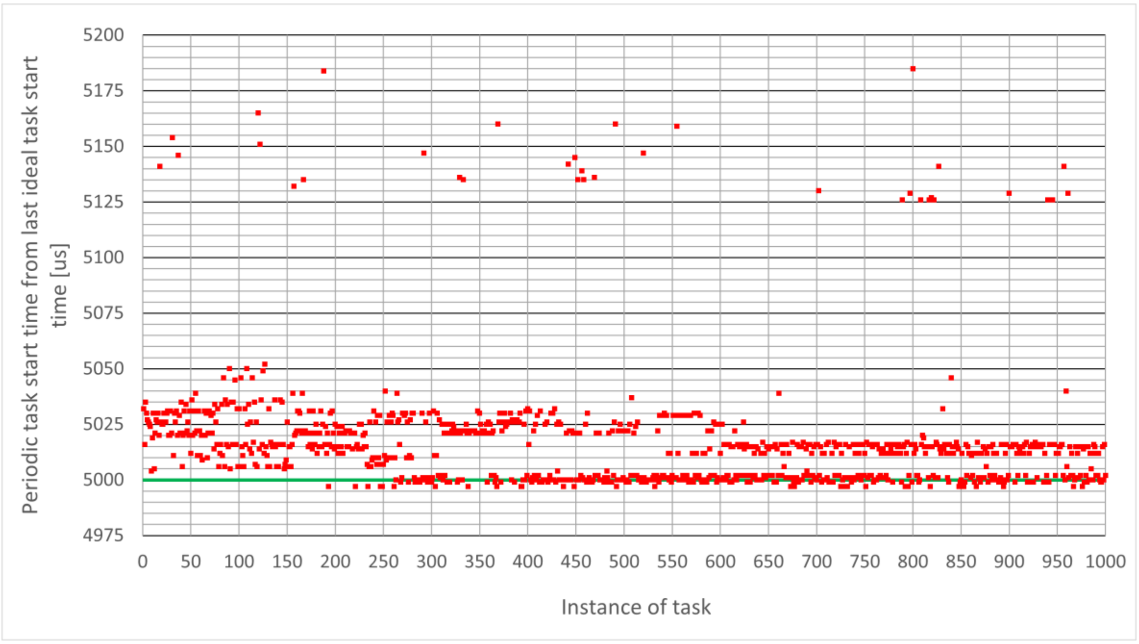


Figure 12. Scheduling jitter of control system – task instance starting time deviations from previous ideal (nominal) task starting time.

4 Real-time operating system implementation

Considering relatively high jitter of control system task, which was found with analysis in last chapter, the implementation of real-time operating system (RTOS) is essential. To find a suitable solution, many RTOSs had to be compared. Following sections are describing the selection process and final implementation on ECU.

4.1 Real-time operating system background

This section gives an understanding of what real-time operating system is and what are the features and functionality it provides.

4.1.1 Embedded systems architecture

An operating system manages different resources of hardware such as memory and processors. A simplified visualisation on Figure 13 shows hardware and software layers of an embedded system. On the left there is a basic system without operating system, but for more complex systems an operating system is typically used as depicted on right side on Figure 13. Operating system manages processor and resources to fulfil the application software. Many embedded systems also have hardware abstraction layer (HAL). This additional layer is between hardware and software layers. It is convenient because porting the application's source code in-between different hardware platforms becomes easier [13].

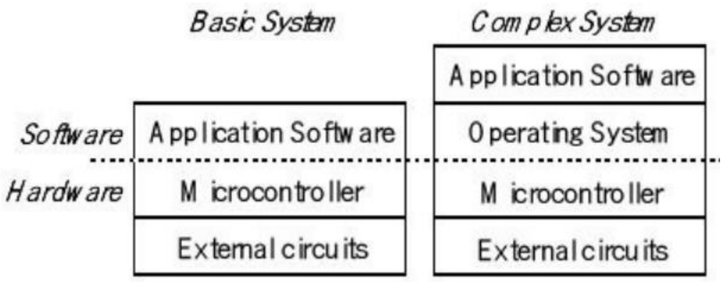


Figure 13. Overview of an embedded system's hardware and software layers [13].

4.1.2 Real-time operating system overview

Real-time operating systems are typically used in applications which have more complex timing requirements. The aim for an RTOS is to meet strict deadlines of task execution in a program in order for time- and safety-critical tasks to be reliable. Critical real-time systems, also referred to as hard real-time systems, have very strict timing deadlines and if these are not met, this results in the system failure and possible damage. Additionally, there are firm and soft real-time systems. Firm real-time systems can allow a few missed deadlines and would not result in a failure right away. Soft real-time systems do not fail completely if deadlines are not met, but the system performance will degrade [14, pp. 40-41].

RTOSs provide execution scheduling which essentially is used for timing of tasks. Widely used scheduling method is priority-based pre-emptive scheduling, which tends to be applied in safety- and time-critical systems. Pre-emption is the ability of scheduler to stop currently running task in order to start handling a task with higher priority. This kind of feature enables the application to always run the most important and urgent task. For pre-emption to work RTOSs also provide various priority levels for tasks [14, p. 42]. These are used to divide tasks into groups based on how critical they are.

In addition to task management and scheduling RTOSs typically provide the following features [15, pp. 143-144]:

- Memory management
- Task communication and synchronisation (with semaphores)
- Timers support
- Interrupt service routines support

A lot of embedded systems' processing units have only one core. This means that there can be only one process running at a time, whereas if a CPU has more cores, it means that many parallel threads can run simultaneously. In simpler systems, functions are typically executed sequentially. However, RTOS's scheduler provides the illusion of multi-tasking even if the system has only one core by rapidly switching between tasks [16]. Following Figure 14 illustrates the multi-tasking behaviour.

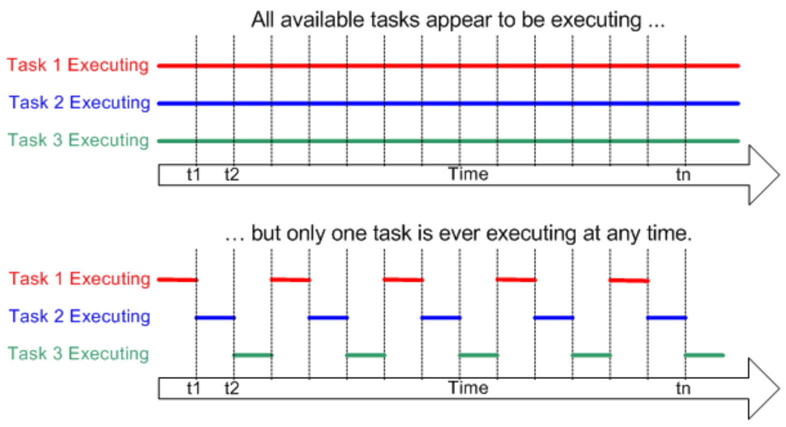


Figure 14. RTOS scheduler rapidly switching between tasks to create multi-tasking effect [17].

4.2 Choice of real-time operating system

Some RTOSs are focused towards different fields and therefore have varying features, making the comparison more complicated. Nevertheless, here are some universal criteria that could be considered [18, pp. 33-34]:

- Memory footprint – RAM and ROM usage
- Performance – context-switching
- Processor architecture support
- Safety-criticality support
- Real-time capability
- Language support
- Documentation, product support
- Debugging tools
- Operating system awareness support in IDE
- Source and object code distribution
- Licensing scheme
- API richness
- Vendor’s reputation

4.2.1 Alternative real-time operating systems

Nowadays many RTOS providers exist who concentrate in various areas like automotive, industrial, IoT (Internet of Things), medical, avionics, consumer electronics. All RTOSs differ in functionality and performance. Some have quicker context-switching, smaller memory footprints and better overall performance, but at the same time lack technical support, documentation or support a very limited amount of processor architectures. Following is a list of some available RTOSs at the moment:

1. FreeRTOS – Amazon Web Services [19]
2. ThreadX (AzureRTOS) – Microsoft, Express logic [20]
3. embOS – SEGGER [21]
4. Keil RTX – ARM [22]
5. SafeRTOS – Wittenstein [23]
6. Zephyr – Zephyr Project [24]
7. QNX Neutrino RTOS – QNX, Blackberry [25]

4.2.2 Comparison of real-time operating systems

From the initial list four most relevant RTOSs were selected mostly based on how much documentation was available. More detailed comparison of these can be seen in Table 1.

Table 1. Comparison table for FreeRTOS, ThreadX, embOS and Keil RTX [18, p. 36], [20], [21], [22], [26], [27], [28], [29], [30], [31].

	FreeRTOS	**ThreadX (AzureRTOS)**	**embOS**	**Keil RTX**
Type	RTOS	RTOS	RTOS	RTOS
Provider (Company)	Amazon Web Services	Microsoft (Express logic)	SEGGER	Arm Limited
License fees	No	No	Yes	No
Domain	Embedded	Embedded	Embedded	Embedded
Language	C	C	C	C
Processor architecture support	ARM Cortex-M (and others)	ARM Cortex-M (and others)	ARM Cortex-M (and others)	ARM Cortex-M (and others)
Compiler support	Arm, GCC, IAR compiler	Arm, GCC, IAR compiler	Arm, GCC, IAR compiler	Arm, GCC, IAR compiler
Atollic TrueStudio kernel awareness	Yes	Yes	Yes	N/A
Scheduler	Pre-emptive, Round-robin, Co-operative	Pre-emptive, Co-operative	Pre-emptive, Round-Robin	Pre-emptive, Round-robin, Co-operative

	FreeRTOS	**ThreadX (AzureRTOS)**	**embOS**	**Keil RTX**
Memory footprint	Kernel 5-10 kB, Scheduler 236 bytes RAM	Instruction area 2 kB, 1 kB RAM	Code space ~1.7 kB, RAM usage < 100 B	Code space < 4 kB, Kernel 428 bytes (RAM)
Context switching	Context switching some microseconds	0.4 µs @ 200MHz	1.5 µs @ 200MHz	2.6 µs @ 72MHz
Safety-criticality support	Possibility to switch to SafeRTOS [23]	Variety of safety-standards	Functional safety certifications	Safe and secure operation
Ease of Use	Simple and intuitive, easy-to-use API	Intuitive	Easy-to-use API	RTX5 aware tools supported in µVision IDE
Documentation, forum support	Very good (online document)	Good (online document)	Good (online document)	Average (online web pages)

From comparison in Table 1 most of the compared RTOSs have similar features and provide all basic functions such as task creation, scheduling based on priority and memory management. They all support C language and GCC (GNU Compiler Collection) compiler, thus can be used with Atollic TrueStudio. In addition, these RTOSs can be used with ECU’s MCU processor architecture. They all support priority based pre-emptive scheduling, which is necessary to fulfil current project’s purpose. Most of the providers claim their RTOS to be easily ported and used. Main differences between these operating systems are memory footprint and task switching times, which affect performance. Memory footprint size is not a huge concern for current project, because memory resources of used MCU are not very limited. The microcontroller used on ECU has 512 kB of SRAM and 2048 kB of flash memory as seen from datasheet [6, p. 17].

It can be seen that embOS [21] is not a free-to-use RTOS, thus this could be excluded from the selection. Keil RTX [22] seems to be most complicated when it comes to porting and implementation, there is not much information on whether Atollic TrueStudio [31] has any awareness support. Additionally, documentation for that RTOS seems to be not that good compared to others. Therefore, Keil RTX is not going to be selected.

ThreadX [20] has good performance, as the context switching times are very small. Provider claims to have good determinism and safety-criticality support as well. Compared to FreeRTOS [19] it tends to be a bit more complex and seems not to have so good forum support. FreeRTOS on the other hand has a very large community, great documentation, and a lot of examples available. Additional advantage is that STM32CubeMX [32] has a built-in support for enabling the FreeRTOS middleware. Based on that the chosen RTOS going to be used on ECU is FreeRTOS.

4.3 Implementation

4.3.1 Task distribution

Most of the tasks in ECU's software are periodic, which means they have certain frequency of execution. The frequencies for each task come from the functionality requirements of that task and are kept the same as they were for previous ECU. With the use of an RTOS, each task can be assigned a priority level. These levels are defined by task's criticality and importance. For example, the *control_system* task is responsible for calculating control system setpoints, thus it is the most important task in ECU and has to occur with the specified frequency for the formula car to perform as expected.

ECU's application code has one aperiodic task which is running in the background most of the time. This task is responsible for processing received information form CAN bus and updating variables. Periodic tasks with their assigned priority levels and execution frequencies can be seen in Table 2.

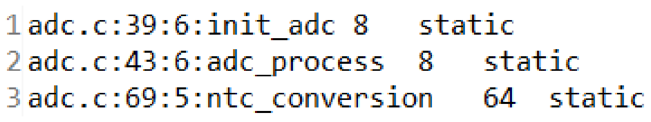
Table 2. Source code periodic tasks' priorities and execution frequencies.

Task name	**Priority**	**Frequency**
control_system	0	200 Hz
heartbeat_sens	1	100 Hz
heartbeat_ams	1	100 Hz
heartbeat_inc	1	100 Hz
drs_control	2	500 Hz
gss_status	2	200 Hz
heartbeat_gss	2	100 Hz
heartbeat_imu	2	100 Hz
tx_can	2	100 Hz
buzzer_control	3	100 Hz
adc_conv	3	100 Hz
pump_control	4	1 Hz
fan_control	4	1 Hz
odometry	4	100 Hz
toggle_led	5	10 Hz

4.3.2 STM32CubeMX configuration

STM32CubeMX is a good tool produced by STMicroelectronics which can be used for configuring the chosen microcontroller. It shows all MCU pins and available functionality under each pin. There is a menu which has all required configuration options, and these can be used to initialise MCU functionality and peripherals as necessary. The software also has clock configuration possibility and middleware configuration such as FreeRTOS. After configuration C code can be generated, which includes drivers, libraries, selected IDE project files and all required initialisation code for peripherals and system [32].

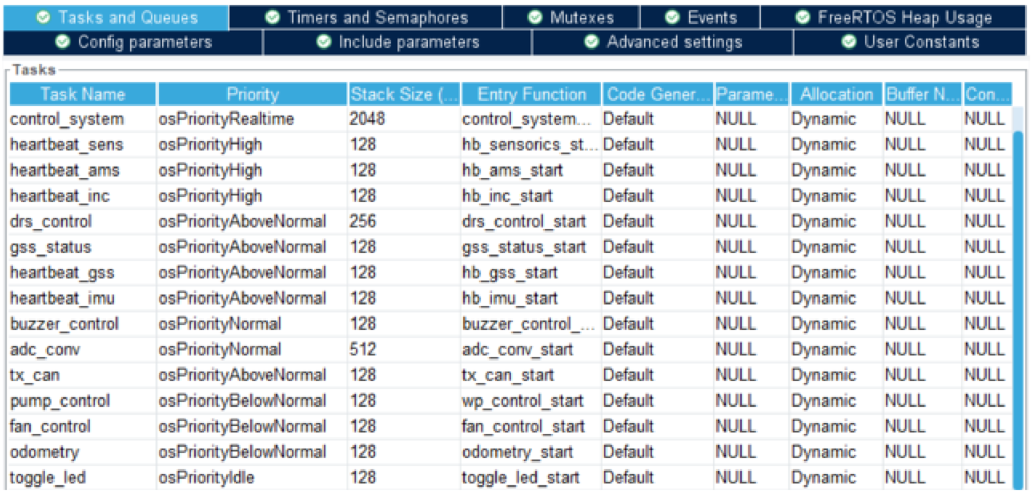
Besides choosing task priorities, configuration of RTOS requires a stack size to be determined for each task. Task's stack size is an important parameter because if it is too small the system can crash. To get some kind of overview of how much stack functions require, a compiler option *-fstack-usage* was added to program code project's compiler settings in Atollic TrueStudio. Now after compiling the project additional files were generated into project's debug folder, which included the information about maximum amount of stack required for each function in the whole project [33]. An example of one such generated file can be seen on Figure 15.



```
adc.c:39:6:init_adc 8   static
adc.c:43:6:adc_process  8   static
adc.c:69:5:ntc_conversion   64  static
```

Figure 15. Generated file containing information about functions stack size.

The number after function name on Figure 15 indicates the required stack size of the corresponding function in bytes. This information was used to figure out initial stack sizes for tasks. Tasks typically include many different functions therefore the stack sizes needed to be summarised to get an indication. Following Figure 16 shows how tasks were configured in STM32CubeMX interface indicating priorities and stack sizes.



Task Name	Priority	Stack Size (...	Entry Function	Code Gener...	Parame...	Allocation	Buffer N...	Con...
control_system	osPriorityRealtime	2048	control_system...	Default	NULL	Dynamic	NULL	NULL
heartbeat_sens	osPriorityHigh	128	hb_sensorics_st...	Default	NULL	Dynamic	NULL	NULL
heartbeat_ams	osPriorityHigh	128	hb_ams_start	Default	NULL	Dynamic	NULL	NULL
heartbeat_inc	osPriorityHigh	128	hb_inc_start	Default	NULL	Dynamic	NULL	NULL
drs_control	osPriorityAboveNormal	256	drs_control_start	Default	NULL	Dynamic	NULL	NULL
gss_status	osPriorityAboveNormal	128	gss_status_start	Default	NULL	Dynamic	NULL	NULL
heartbeat_gss	osPriorityAboveNormal	128	hb_gss_start	Default	NULL	Dynamic	NULL	NULL
heartbeat_imu	osPriorityAboveNormal	128	hb_imu_start	Default	NULL	Dynamic	NULL	NULL
buzzer_control	osPriorityNormal	128	buzzer_control_...	Default	NULL	Dynamic	NULL	NULL
adc_conv	osPriorityNormal	512	adc_conv_start	Default	NULL	Dynamic	NULL	NULL
tx_can	osPriorityAboveNormal	128	tx_can_start	Default	NULL	Dynamic	NULL	NULL
pump_control	osPriorityBelowNormal	128	wp_control_start	Default	NULL	Dynamic	NULL	NULL
fan_control	osPriorityBelowNormal	128	fan_control_start	Default	NULL	Dynamic	NULL	NULL
odometry	osPriorityBelowNormal	128	odometry_start	Default	NULL	Dynamic	NULL	NULL
toggle_led	osPriorityIdle	128	toggle_led_start	Default	NULL	Dynamic	NULL	NULL

Figure 16. RTOS tasks configuration view in STM32CubeMX.

4.3.3 Timing of tasks

Most of the tasks in ECU's source code have periodic nature, it means they are called with a certain interval. A default FreeRTOS task is idle and therefore running continuously in the background only to be pre-empted by another higher priority task. To achieve periodic execution, FreeRTOS API provides delay functions such as *vTaskDelayUntil()* and *vTaskDelay()*. Difference between these functions is the way delay is specified. *vTaskDelay()* creates a relative delay from the moment the function itself is called, but *vTaskDelayUntil()* creates an absolute delay time after which task can unblock and execute. This means using *vTaskDelayUntil()* function establishes a constant execution rate for a periodic task and interruptions or pre-emption cannot alter time period length between tasks as it could be with *vTaskDelay()* [19, pp. 48-51].

For this reason timing of periodic tasks in ECU's source code is done using *vTaskDelayUntil()* function which could contribute to achieving smaller jitter. Example of how this function is used in one of ECU's tasks is shown on Figure 17.

```
void adc_conv_start(void const * argument)
{
      /* USER CODE BEGIN adc_conv_start */
      TickType_t xLastWakeTime;
      const TickType_t xFrequency = 10;      // 100 Hz
      xLastWakeTime = xTaskGetTickCount();
      /* Infinite loop */
      for(;;)
      {
            vTaskDelayUntil(&xLastWakeTime, xFrequency);
            adc_process();
            osDelay(1);
      }
      /* USER CODE END adc_conv_start */
}
```

Figure 17. Example of timing of periodic tasks.

4.3.4 FreeRTOS scheduling overview

FreeRTOS has various task states which are used for scheduling tasks. Each task can be in either of the following states [27, pp. 65-66]:

- **Running state** – task being in the running state is the task, which is currently executed, meaning it uses the processor.
- **Blocked state** – task being in the blocked state is currently not running and is waiting for an event to happen to get into ready state. The event can be for example an absolute delay period ending. Blocked tasks also have a timeout period after which they are unblocked to move into ready state.
- **Suspended state** – task being in suspended state is currently not running and also not available for the scheduler. Task can only enter and exit suspended state with certain function calls.
- **Ready state** – task being in a ready state is not currently running but is ready to begin execution waiting for scheduler.

These task states with possible transitions can be seen on Figure 18. Majority of tasks in ECU are scheduled by using a delay function as covered in previous section. The *vTaskDelayUntil()* function puts a task in blocked state after it has finished its execution

[19, p. 50]. The task waits in blocked state until delay time period expires and then transitions into ready state to be executed again. This cycle is depicted on Figure 18 with bold lines.

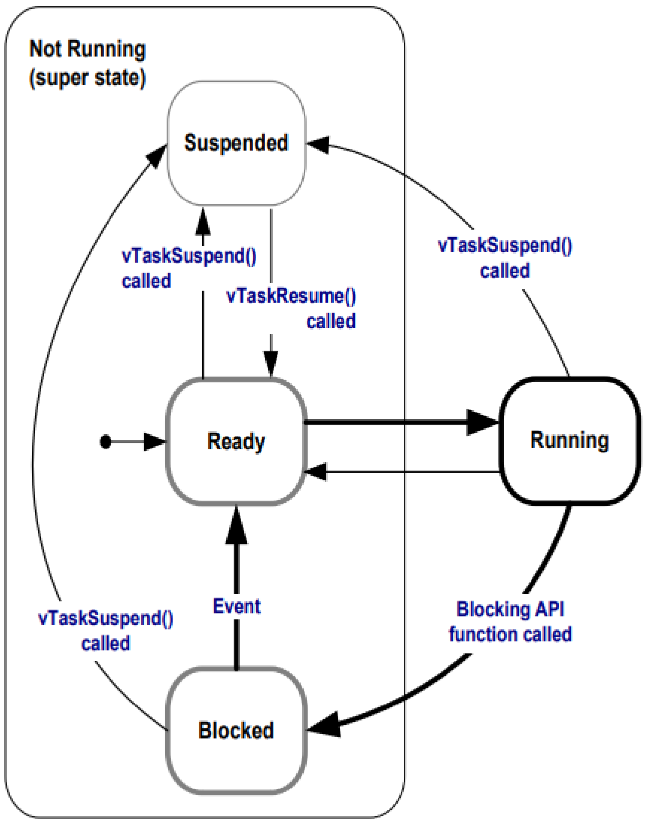


Figure 18. Real-time operating system possible task states and transitions between them [27, p. 71].

5 Control System modifications

All of the control system input variables are essential to calculate output setpoints which are sent to motor controller. The input variables are updated in ECU before every control system task execution and all the calculations are performed during one iteration. Simulink model of control system has many different sections that serve individual purpose as seen on Figure 3. Sections divide into blocks which are interconnected by different signals that essentially are the variables used in the whole model. Blocks contain function-blocks and formulas, which use the signal variables.

FEST20 control system uses a bus system in Simulink model, which combines many separate signals into different buses [1]. After generating C language source code these signal buses appear in code as structures containing Simulink model signals as data variables. Some structures contain other structures inside of them. All structures consisting of setpoints data contain this data for every motor. Additionally, Embedded Coder generates many auxiliary variables, which are used in the process of calculating setpoints.

Majority of used variables are of type *double*, which is a 64-bit data type. This data type has a floating-point format. Another floating-point format data type is known as *float,* but this type stores 32 bits [34]. Compared to *double* type *float* takes two times less memory space. It has single precision whereas *double* has double precision which means that a variable of type *double* could have two times more precision. On the other hand, processing of *float* type variables in a program takes less time, which makes the whole application run faster [35, p. 24].

Additionally, there is the *int* data type, which has fixed-point format. This is one of the most basic data types used in embedded systems software. This type stores 32-bit integer. Processing *int* data types is also faster than processing *double* because operations take less CPU cycles [35, p. 6]. Following sections are covering information about using *int* or *float* types instead of *double* to improve control system task execution times.

5.1 Floating-point unit

STM32F7 microcontrollers host an FPU which is implemented in the MCU’s hardware. FPU helps to improve performance when it comes to computing with floating-point variables such as *float* or *double*. Some STMicroelectronics’ MCUs have only SP-FPU meaning that calculations with *float* data type could be made quicker. But to also gain calculation speed with *double* variables, hardware DP-FPU is needed. There are alternative software algorithms which try to imitate hardware FPU, but these do not have the same effect on performance, because hardware implementation does operations with less CPU cycles [35].

5.1.1 Julia set testing

Julia set is a complementary set, which can be used to calculate a mathematical fractal [35, p. 18]. Data types used for calculation can be changed and therefore this set can be used to compare MCU’s performance with different data types. Julia set generation function can be seen on Appendix 2. For testing purposes this function has been slightly modified. Definitions *IMG_CONSTANT* and *REAL_CONSTANT* have been replaced with a floating-point number and the data buffer is declared inside the function. These modifications do not affect the outcome in this sense, that this test is only about how long it takes for the MCU to execute one iteration of this function. Modified Julia set function can be seen on Figure 19.

```
void GenerateJulia_fpu(uint16_t size_x, uint16_t size_y, uint16_t
offset_x, uint16_t offset_y, uint16_t zoom)
{
	float tmp1, tmp2;
	float num_real, num_img;
	float radius;
	uint8_t i;
	uint16_t x,y;
	uint8_t buffer[115680];
	for (y=0; y<size_y; y++)
	{
		for (x=0; x<size_x; x++)
		{
			num_real = y - offset_y;
			num_real = num_real / zoom;
			num_img = x - offset_x;
			num_img = num_img / zoom;
			i=0;
			radius = 0;
			while ((i<ITERATION-1) && (radius < 4))
			{
				tmp1 = num_real * num_real;
				tmp2 = num_img * num_img;
				num_img = 2*num_real*num_img + 3.14;
				num_real = tmp1 - tmp2 + 3.14;
				radius = tmp1 + tmp2;
				i++;
			}
			buffer[x+y*size_x] = i;
		}
	}
	HAL_GPIO_TogglePin(LED1_GPIO_Port, LED1_Pin);
}
```

Figure 19. Modified Julia set function used to measure execution times with different data types.

The *GenerateJulia_fpu()* function is called in the endless while loop and there are no more functions in the loop. An LED toggling function has been added to the end of the Julia set function. This means every time the function completes, an LED on ECU changes its state. Voltage on LED’s anode terminal has been measured with oscilloscope to determine the time differences between LED state changes. Plot on Figure 20 shows oscilloscope display view of LED states measurement. That measurement corresponds to the test set, which used *float* type.

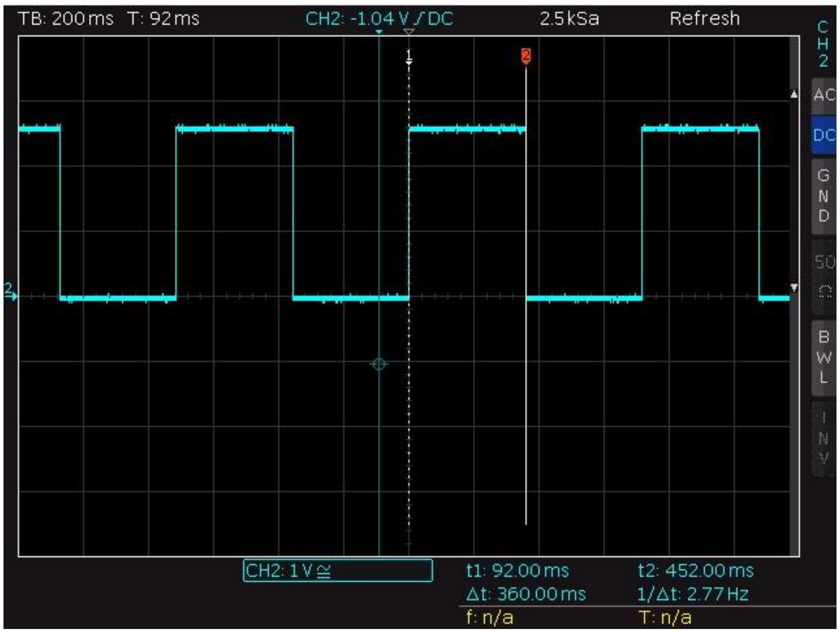


Figure 20. LED state changes after each Julia set calculation seen on oscilloscope display.

Six different test sets were carried out. In these sets DP-FPU was used with *double*, *float* and *uint32_t* types, then SP-FPU with *float* and *uint32_t* and lastly hardware FPU disabled with *uint32_t* data type. Activation of FPU was changed from compiler configuration options. Used data types were changed for variables *tmp1*, *tmp2*, *num_real*, *num_img* and *radius*. Results are shown in Table 3. As seen even if using DP-FPU to calculate *double* types, it takes still the longest to execute the whole function. On the other hand, using *float* has considerable improvement in execution time. Using *uint32_t* gives further improvements so this could be the fastest, but with integers there is a greater loss in precision.

Table 3. Test results of modified Julia set calculation times.

Test no.	**FPU**	**Data type**	**Execution time (ms)**
Test 1	DP-FPU	*double*	424
Test 2	DP-FPU	*float*	360
Test 3	DP-FPU	*uint32_t*	330
Test 4	SP-FPU	*float*	360
Test 5	SP-FPU	*uint32_t*	330
Test 6	None	*uint32_t*	330

5.2 Control system model optimisation

One way to improve control system task execution times would be to optimise the used data types in the model. The tests carried out in last section using Julia set calculation function [35] showed promising results with *float* and *uint32_t* data types providing significantly quicker execution. Based on that the *double* data types used for control system signals are replaced.

Control system has used many *double* variables in the process of calculating setpoints. After setpoint data is ready, ECU passes these variables on to functions which are responsible for storing them into CAN transmission mailboxes. Mailboxes store up to three messages before transmitting them onto CAN bus. Standard CAN message contains 64 bits or 8 bytes of payload data [36, p. 13]. These bytes are transmitted as integers which means that the setpoint variables are casted from *double* type to either unsigned or signed integer. This also means that the calculated values are rounded.

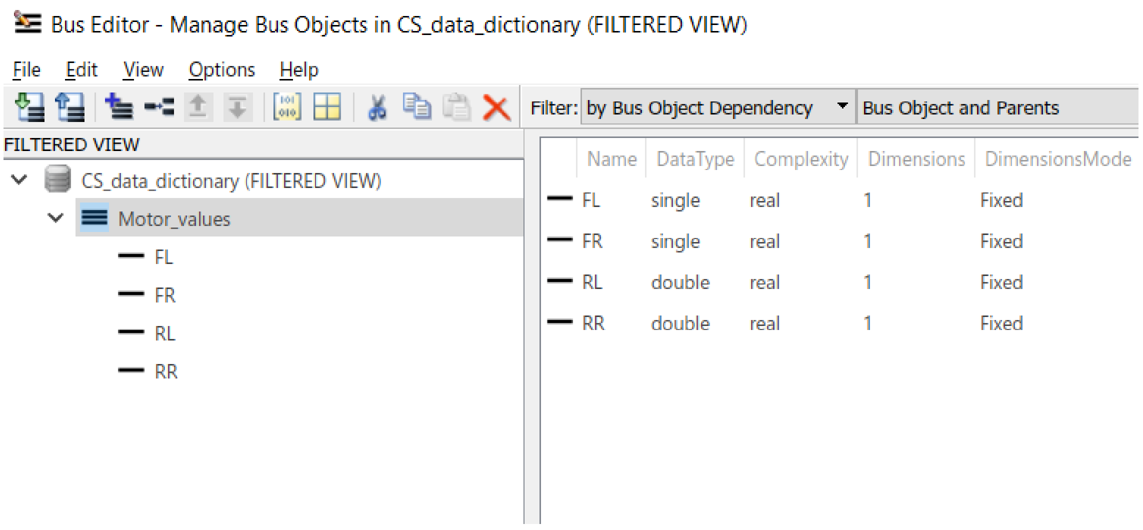
Therefore it could be possible to even use integer type variables in control system model which would probably make the best performance. Disadvantage comes from the fact that if integers are used for a lot of consecutive calculations the end result may differ quite a bit because these variables do not have decimal point. Using integers would require a system, where every variable is multiplied by powers of 10 during calculations and the end result is divided accordingly to gain in precision. Easier would be to use *float* types, which have single-precision and would provide more precise end results for consecutive calculations. Additionally the task execution time difference from testing results in Table 3 between *uint32_t* and *float* is not so remarkable.

5.2.1 Editing data types in Simulink

Simulink environment has a tool called bus editor. This editor lists all buses which are used in the project and indicates the signals which are inside of those buses. Parameters of buses and signals can be configured easily by using this interface. An example on Figure 21 is shown where data types of two signals are changed from *double* into *single* (*float*).

It is better for other data types and block outputs in the model to have inherited data type, because this can automatically assign their type based on input. Furthermore, in

Embedded coder under code generation settings, there is an option to choose a default data type for underspecified type signals to be of type *single*.


Bus Editor - Manage Bus Objects in CS_data_dictionary (FILTERED VIEW)
File Edit View Options Help
Filter: by Bus Object Dependency
Bus Object and Parents
FILTERED VIEW
CS_data_dictionary (FILTERED VIEW)
Motor_values
FL
FR
RL
RR

Name	DataType	Complexity	Dimensions	DimensionsMode
FL	single	real	1	Fixed
FR	single	real	1	Fixed
RL	double	real	1	Fixed
RR	double	real	1	Fixed

Figure 21. Signal variables data type configuration in Simulink bus editor.

6 Analysis of new solution

This section is covering the analysis of the new ECU solution. Its program code is based on RTOS and control system data types are optimised. The ECU hardware platform has irrelevant changes in case of testing which do not affect the performance. Formula car in which the ECU is tested is the same as previously.

6.1 Results achieved

Same data was gathered for newly implemented ECU to be able to compare both solutions and find whether new solution has advanced as expected.

6.1.1 Task execution time of optimised control system

This analysis was done with control system using *float* data types. Other tasks are added together and depicted as one grey column. Execution times of other tasks are similar to previous analysis on Figure 9. This seems correct as these tasks were not optimised. As seen from Figure 22 worst-case execution time for whole program iteration could be 1252 μs. This corresponds to approximately 799 Hz of iteration computation frequency, which is remarkably better compared to calculation frequency with non-optimised control system. This result leaves plenty of reserve for control system task to be executed with 500 Hz. Additionally, from Figure 22 it seems that the deviation between minimum and maximum execution times has reduced, which means more consistent execution.

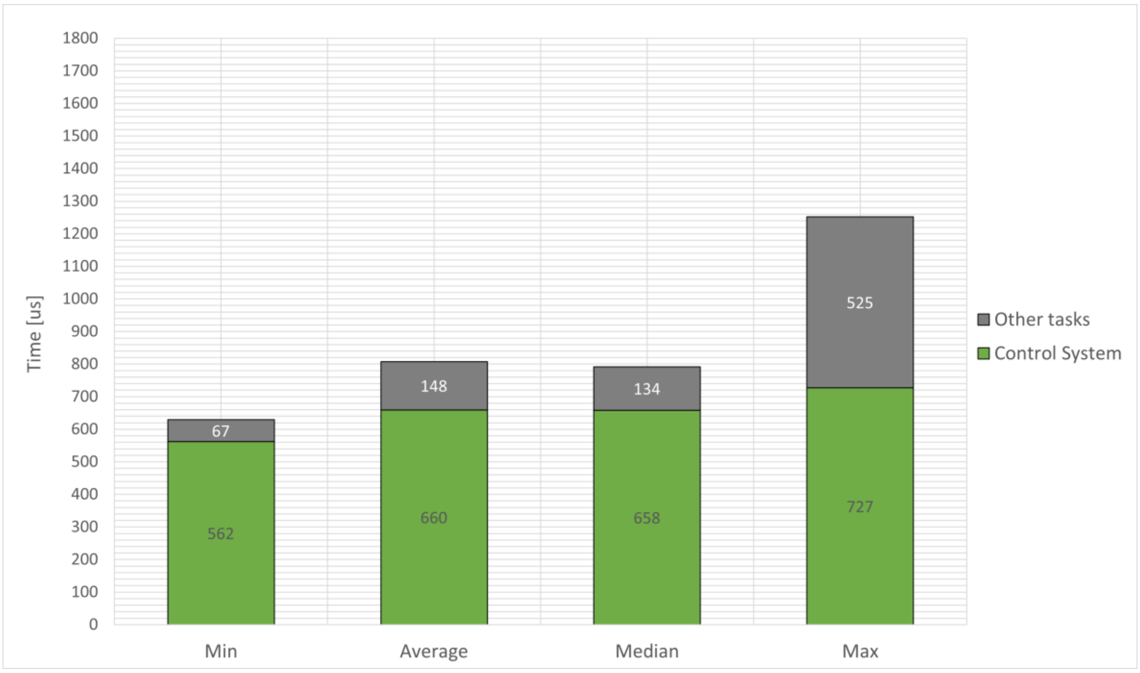


Figure 22. Task execution times with optimised control system.

6.1.2 WCET analysis of optimised control system

WCET analysis was done based on similar amount of sample execution occurrences as with previous system. Plot on Figure 23 gives overview of the distribution of execution times. Again it can be seen that execution times are more concentrated towards the average, so difference from minimum to maximum execution time is smaller with the new solution. There is no distinctive lower range of running times. Average execution time range of control systems is about 635 – 685 μs, this is almost the same as before. BCET for control system is 562 μs and WCET is 727 μs, which is significantly better than the previous system without RTOS and using *double* types.

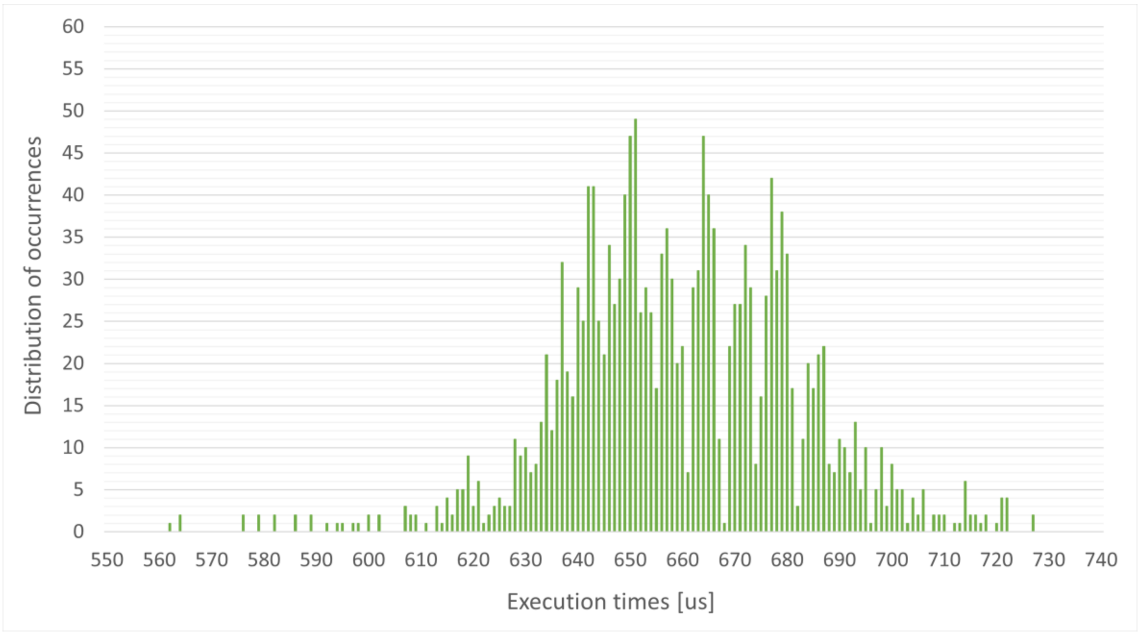


Figure 23. Control system execution times using float data type.

6.1.3 Scheduling jitter of control system with RTOS

Plot on Figure 24 visualises scheduling jitter analysis of 1000 control system task instances. Vertical axis upper and lower bounds are kept the same as used previously on Figure 12 to better indicate the difference that occurs when control system task is scheduled by an RTOS. Graph on Figure 24 shows that most task instances have minor jitter and are quite close to the green line indicating ideal (nominal) execution period ranging from -5 to 5 μs. Some instances are under the nominal line in the range of -12 to -8 μs, this means that in those cases tasks execute a little bit earlier than ideal, which does not create a problem as the differences are very small. Maximum deviation which goes over the nominal execution period time is 13 μs and biggest deviation under nominal is -14 μs. Accordingly the worst-case jitter for control system task execution with RTOS implemented is 27 μs which is notably better in contrast with previously used timing system not employing RTOS.

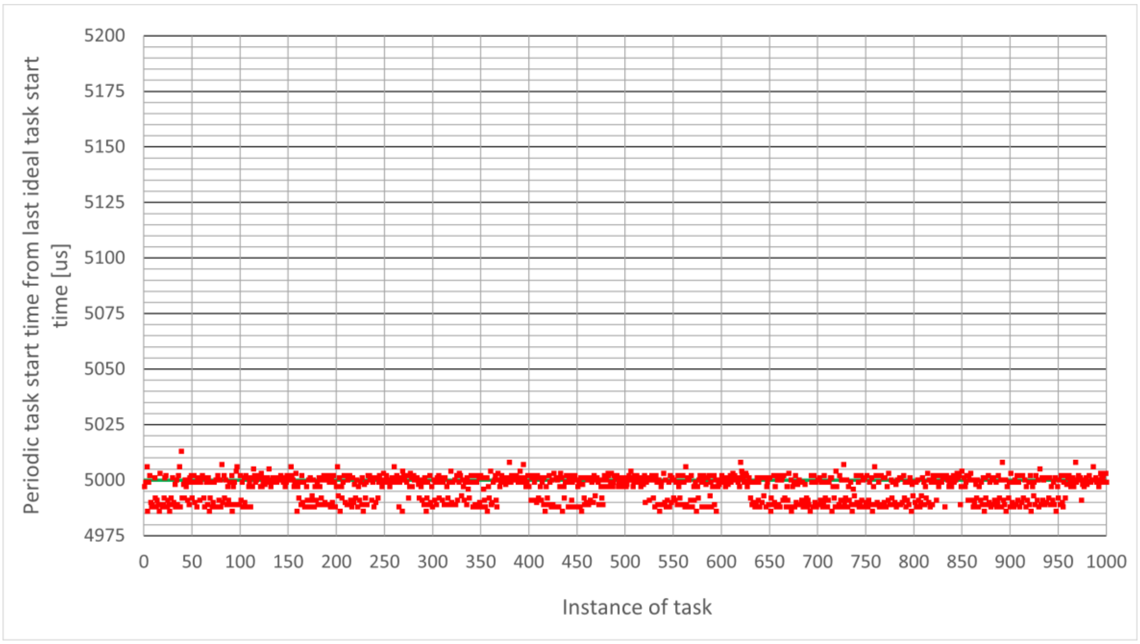


Figure 24. Scheduling jitter of control system with RTOS – task instance starting time deviations from previous ideal (nominal) task starting time.

6.2 Comparison of both solutions

This section brings out concrete differences between the two ECU solutions. Firstly task execution time of control system is compared, following is the difference between control system task scheduling jitter.

6.2.1 Control system task execution time comparison

Measurements were carried out with using both *double* and *float* data types in otherwise similar conditions. Testing set using *double* was the same as for previous system testing, but now RTOS was used for scheduling. Interestingly, the deviation between minimum and maximum running times was smaller even for *double* types. Similar behaviour goes for *float* testing set. Table 4 provides speed increase percentage from *double* to *float* and it is safe to say that this optimisation had valuable effects on the performance of ECU.

Table 4. Control system execution times with using *double* or *float* comparison.

	Execution time [ms]			
	Minimum	**Average**	**Median**	**Maximum**
double	931	1049	1034	1193
float	562	660	658	727
Speed increase	39.63%	37.12%	36.36%	39.06%

6.2.2 Scheduling jitter comparison

Finally previously covered scheduling jitter analysis results have been depicted on the same graph seen on Figure 25. RTOS scheduling is indicated with green dots and self-developed variant with red dots. With its small deviations RTOS scheduling seems a lot more consistent. It tends to start tasks little bit before the nominal period from last ideal execution, whereas self-developed timing system executes tasks after deadline. Self-developed system has more variations in individual task's jitter and even some serious deviations which should not be present in a hard real-time system. RTOS scheduling clearly has smaller worst-case jitter.

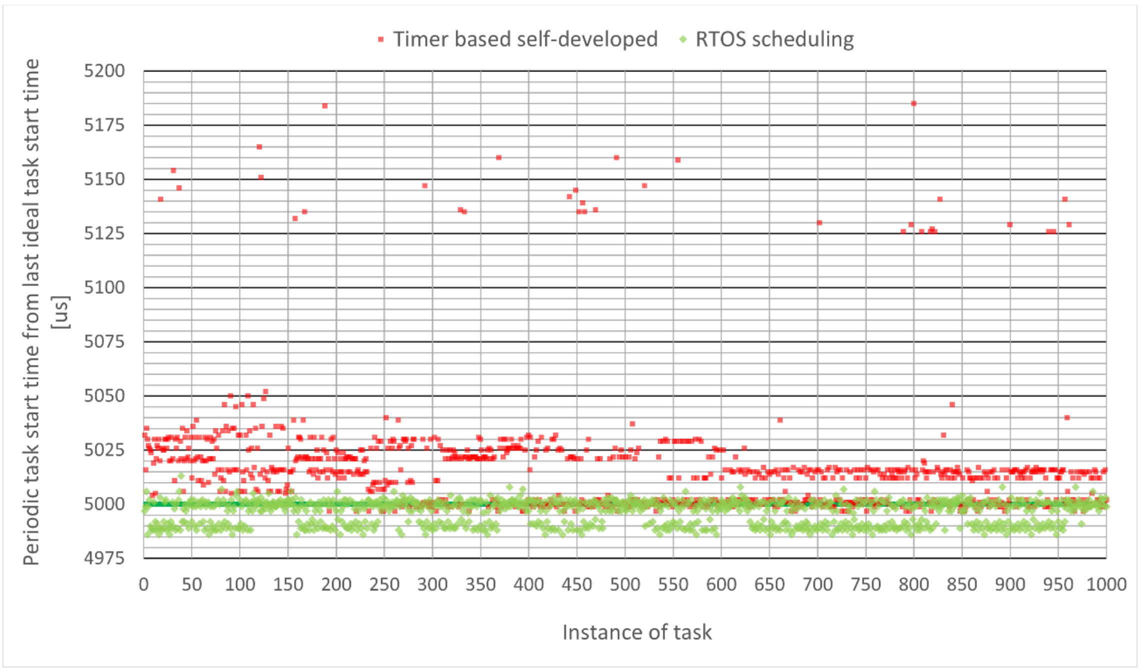


Figure 25. Control system scheduling jitter – self-developed vs RTOS scheduling.

7 Summary

Current thesis was presented in order to develop enhanced version of Electronic Control Unit which was used for controlling the motors of Formula Student Team Tallinn’s formula car FEST20. The software of previous ECU lacked time-criticality and ability to run tasks concurrently based on their priorities. Furthermore, emerged interest in raising control system task execution rate for testing purposes was complicated with previous ECU because of relatively long and varying control system execution times.

This work tackles with providing solutions to aforementioned problems. Firstly previous ECU is analysed to determine system performance and define problematic areas. Next sections introduce solutions which are the basis for developing new improved ECU solution. In the end comparison between both ECU solutions is conducted to underline benefits gained from the new system and get an overview of system’s performance.

Previously used self-developed timing system is replaced by a real-time operating system, which enables prioritisation of tasks and pre-emptive scheduling. It refines time-criticality of the whole system. To achieve greater control system execution rates, the control system model is optimised. Analysis of both ECU solutions determine task execution times, WCET and scheduling jitter.

As a result of this thesis a refined version of ECU has been implemented. Important and critical tasks have higher priorities and can pre-empt other less important tasks. Task scheduling jitter is smaller thus providing more reliable operation. Control system execution times are shorter, meaning higher computation rate could be tested.

Further work could be to decrease the deviation between minimum and maximum execution times of control system to improve system determinism and predictability. Additionally higher execution rate tests can be carried out.

References

[1] Formula Student Team Tallinn, “Control System 2020,” 2020. [Online]. Available: https://conf.formulastudent.ee/display/FEST20/20.02.01.0+Control+System. [Accessed 01.05.2021].

[2] Formula Student Germany GmbH, “Formula Student Rules 2020,” 2020. [Online]. Available: https://www.formulastudent.de/fileadmin/user_upload/all/2020/rules/FS-Rules_2020_V1.0.pdf. [Accessed 27.04.2021].

[3] Formula Student Team Tallinn, *Eesti tudengid ehitavad uued elektrivormelid ka tänavu*, 2021. [Online]. Available: https://www.formulastudent.ee/eesti-tudengid-ehitavad-uued-elektrivormelid-ka-tanavu/. [Accessed 03.05.2021].

[4] O. Avatefipour and H. Malik, “State-of-the-Art Survey on In-Vehicle Network Communication "CAN-Bus" Security and Vulnerabilities,” *International Journal of Computer Science and Network*, vol. 6, no. 6, pp. 720-727, 2017. [Online]. Available: https://www.researchgate.net/publication/321474822_State-of-the-Art_Survey_on_In-Vehicle_Network_Communication_CAN-Bus_Security_and_Vulnerabilities. [Accessed 01.05.2021].

[5] Formula Student Team Tallinn, “Electronic Control Unit Design,” 2020. [Online]. Available: https://conf.formulastudent.ee/display/FEST20/Electronic+Control+Unit+Design. [Accessed 05.05.2021].

[6] *STM32F767VI, Microcontroller*, Datasheet, STMicroelectronics, 2021. [Online]. Available: https://www.st.com/resource/en/datasheet/stm32f767vi.pdf. [Accessed 30.04.2021].

[7] Simulink. 2021, The MathWorks. [Online]. Available: https://www.mathworks.com/help/simulink/index.html. [Accessed 01.05.2021].

[8] Embedded Coder. 2021, The MathWorks. [Online]. Available: https://www.mathworks.com/help/ecoder/. [Accessed 01.05.2021].

[9] J. Engblom, A. Ermedahl, M. Sjdin, et al., “Execution-time analysis for embedded real-time systems,” *International Journal on Software Tools for Technology Transfer,* 2001. [Online]. Available: https://www.researchgate.net/publication/246128205_Execution-time_analysis_for_embedded_real-time_systems. [Accessed 04.05.2021].

[10] P. Moryc and J. Černohorský, “Task jitter measurement under RTLinux operating system,” in *Proceedings of the International Multiconference on Computer Science and Information Technology*, Ostrava, Technical University of Ostrava, 2008, p. 849 – 858. [Online]. Available: https://annals-csis.org/proceedings/2007/pliks/48.pdf. [Accessed 07.05.2021].

[11] F. M. Proctor and W. P. Shackleford, “Real-time Operating System Timing Jitter and its Impact on Motor Control,” *Proceedings of the SPIE Sensors and Controls for Intelligent Manufacturing II,* vol. 4563, pp. 10-16, 2001, doi: 10.1117/12.452653.

[12] HMO3524, 350 MHz Mixed Signal Oscilloscope, User Manual, HAMEG Instruments GmbH, 2013. [Online]. Available: https://cdn.rohde-schwarz.com/hameg-archive/HAMEG_MAN_EN_HMO3524_.pdf. [Accessed 30.04.2021].

[13] J. W. Valvano, Embedded Systems: RealTime Operating Systems for ARM® Cortex-M Microcontrollers, 4 ed., vol. 3, J. W. Valvano, 2017. http://users.ece.utexas.edu/~valvano/, ISBN: 978-1466468863.

[14] W. Cedeño and P. A. Laplante, “An Overview of Real-Time Operating Systems,” *Journal of the Association for Laboratory Automation,* vol. 12, pp. 40-45, 2007, doi: 10.1016/j.jala.2006.10.016.

[15] S. Iyer and P. Gupta, Embedded Realtime Systems Programming, New Delhi: Tata McGraw-Hill, 2003. [Online]. Available: https://books.google.co.bw/books?id=t4siBAAAQBAJ. [Accessed 05.05.2021].

[16] FreeRTOS, “What is An RTOS?,” Amazon Web Services, 2021. [Online]. Available: https://www.freertos.org/about-RTOS.html. [Accessed 05.05.2021].

[17] FreeRTOS, “Multitasking,” Amazon Web Services, 2021. [Online]. Available: https://www.freertos.org/implementation/a00004.html. [Accessed 05.05.2021].

[18] T. N. B. Anh and S.-L. Tan, “Real-Time Operating Systems for Small Microcontrollers,” *IEEE Micro,* vol. 29, no. 5, pp. 30-45, 2009, doi: 10.1109/MM.2009.86.

[19] FreeRTOS, Real-Time Operating System, Reference Manual, Amazon Web Services, 2017. [Online]. Available: https://www.freertos.org/fr-content-src/uploads/2018/07/FreeRTOS_Reference_Manual_V10.0.0.pdf. [Accessed 03.05.2021].

[20] Express Logic, “Azure RTOS, What is Microsoft Azure RTOS?,” Microsoft, 2021. [Online]. Available: https://docs.microsoft.com/en-us/azure/rtos/overview-rtos. [Accessed 05.05.2021].

[21] embOS, Real-Time Operating System, Reference Manual, SEGGER Microcontroller GmbH, Monheim am Rhein, 2020. [Online]. Available: https://www.segger.com/doc/UM01001_embOS.html. [Accessed 05.05.2021].

[22] Arm Limited, “Keil RTX5,” Arm Limited, 2019. [Online]. Available: https://www2.keil.com/mdk5/cmsis/rtx. [Accessed 05.05.2021].

[23] SafeRTOS, Real-Time Operating System, User Manual, WITTENSTEIN aerospace & simulation, 2020. [Online]. Available: https://www.highintegritysystems.com/downloads/manuals_and_datasheets/Sample_SafeRTOS_User_Manual.pdf. [Accessed 05.05.2021].

[24] Zephyr Project, “Introduction,” Linux Foundation, 04 05 2021. [Online]. Available: https://docs.zephyrproject.org/latest/introduction/index.html. [Accessed 05.05.2021].

[25] QNX Software Systems, “About the QNX Neutrino User's Guide,” QNX Software Systems, 2020. [Online]. Available:

http://www.qnx.com/developers/docs/7.1/index.html#com.qnx.doc.neutrino.
user_guide/topic/about.html. [Accessed 05.05.2021].

[26] Arm Limited, “CMSIS-RTOS2 Documentation,” Arm Limited, 2020. [Online]. Available: https://arm-software.github.io/CMSIS_5/RTOS2/html/index.html. [Accessed 05.05.2021].

[27] R. Barry, “Mastering the FreeRTOS™ Real Time Kernel,” Real Time Engineers Ltd, 2016. [Online]. Available: https://www.freertos.org/fr-content-src/uploads/2018/07/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf. [Accessed 05.05.2021].

[28] RL-ARM (MDK v4), User Guide, Arm Limited, 2019. [Online]. Available: https://www.keil.com/support/man/docs/rlarm/rlarm_ar_artxarm.htm. [Accessed 05.05.2021].

[29] Azure RTOS ThreadX, Real-Time Operating System, Documentation, Microsoft, 2020. [Online]. Available: https://opdhsblobprod04.blob.core.windows.net/contents/bbbb93dd48c746e19c79a6460680d524/59c5cbfbb3a3868b42e05fe3be844838?sv=2018-03-28&sr=b&si=ReadPolicy&sig=VfVgM6N04D%2B30RjReuAr2LPBPC9BVJFSrwsgoRphMfc%3D&st=2021-05-06T09%3A22%3A41Z&se=2021-05-07T09%3A3. [Accessed 05.05.2021].

[30] FreeRTOS, “FreeRTOS FAQ - Memory Usage, Boot Times & Context Switch Times,” Amazon Web Services, 2021. [Online]. Available: https://freertos.org/FAQMem.html#ContextSwitchTime. [Accessed 05.05.2021].

[31] Atollic TrueSTUDIO, Integrated Development Environment, User Guide, STMicroelectronics, Jönköping, 2018. [Online]. Available: https://www.st.com/resource/en/user_manual/usermanual_truestudio-user-manual-for-truestudio-930-stmicroelectronics.pdf. [Accessed 05.05.2021].

[32] STMicroelectronics, “STM32CubeMX, STM32Cube initialization code generator,” STMicroelectronics, 2021. [Online]. Available: https://www.st.com/en/development-tools/stm32cubemx.html. [Accessed 05.05.2021].

[33] AdaCore, “GNAT User’s Guide for Native Platforms,” Free Software Foundation, 2021. [Online]. Available: https://gcc.gnu.org/onlinedocs/gnat_ugn/Static-Stack-Usage-Analysis.html#Static-Stack-Usage-Analysis. [Accessed 04.05.2021].

[34] Sun Microsystems, Inc, “Numerical Computation Guide Chapter 2,” [Online]. Available: https://docs.oracle.com/cd/E19957-01/806-3568/ncg_math.html. [Accessed 03.05.2021].

[35] STMicroelectronics, “Floating point unit demonstration on STM32 microcontrollers,” May 2016. [Online]. Available: https://www.st.com/resource/en/application_note/dm00047230-floating-point-unit-demonstration-on-stm32-microcontrollers-stmicroelectronics.pdf. [Accessed 03.05.2021].

[36] Robert Bosch GmbH, “CAN Specification Version 2.0,” Robert Bosch GmbH, Stuttgart, 1991.

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Rain Sarapuu

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Performance Improvement of Electronic Control Unit for Formula Student Electric Car”, supervised by Mairo Leier
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

10.05.2021

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 – C language function to calculate Julia set

This function snippet originates from STMicroelectronics' document [35].

```c
void GenerateJulia_fpu(uint16_t size_x, uint16_t size_y, uint16_t
offset_x, uint16_t offset_y, uint16_t zoom, uint8_t * buffer)
{
      double        tmp1, tmp2;
      double        num_real, num_img;
      double        radius;
      uint8_t            i;
      uint16_t         x,y;

      for (y=0; y<size_y; y++)
      {
            for (x=0; x<size_x; x++)
            {
                  num_real = y - offset_y;
                  num_real = num_real / zoom;
                  num_img = x - offset_x;
                  num_img = num_img / zoom;
                  i=0;
                  radius = 0;
            while ((i<ITERATION-1) && (radius < 4))
                  {
                        tmp1 = num_real * num_real;
                        tmp2 = num_img * num_img;
                        num_img = 2*num_real*num_img + IMG_CONSTANT;
                        num_real = tmp1 - tmp2 + REAL_CONSTANT;
                        radius = tmp1 + tmp2;
                        i++;
                  }
                  /* Store the value in the buffer */
                  buffer[x+y*size_x] = i;
            }
    }
}
```