Modern C++ and Rust in embedded memory-constrained systems

Master's thesis in Embedded Electronic System Design

Ashwin Kumar Balakrishnan

Gaurav Nattanmai Ganesh

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Modern C++ and Rust in embedded memory-constrained systems

Ashwin Kumar Balakrishnan
Gaurav Nattanmai Ganesh

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Ashwin Kumar Balakrishnan
Gaurav Nattanmai Ganesh

Supervisor: Lars Svensson, Department of Computer Science and Engineering
Advisor: Peter Lööf, QRTECH AB
Examiner: Per Larsson Edefors, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Description of the picture on the cover page (if applicable)

Typeset in LaTeX
Gothenburg, Sweden 2022

Ashwin Kumar Balakrishnan
Gaurav Nattanmai Ganesh
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Low level languages like C and traditional C++ have been used extensively in embedded systems for a long time due to the critical advantages such as low runtime overhead and good memory management, despite their memory safety issues. In an attempt to overcome the safety issues, many high level languages have been introduced recently, albeit with a high execution time overhead. This thesis mainly focuses on Modern C++ and Rust, which is a multi-paradigm language introduced for memory safety and improved performance. We make a comparison study on using these two high level languages in a memory-constrained embedded system.

The comparison is made by running both the languages in resource intensive applications, such as artificial engine sound generation and Quicksort of large arrays on a single hardware platform. The performance analysis focuses on the parameters: execution time, memory usage and development time, which develops a conclusion as to why high level languages can be used in memory constrained embedded systems and which language performs better in this case.

Acknowledgements

We would like to express our heartfelt thanks to our supervisor Lars Svensson and Examiner Per Larsson-Edefors for helping us with their guidance during our thesis work.
We would also like to thank the people at Qrtech AB for providing us the opportunity to do thesis work at their company, and especially Peter Lööf for his continual guidance and suggestions to make the thesis better.

Ashwin Kumar Balakrishnan & Gaurav Nattanmai Ganesh, Gothenburg, January 2022

Contents

List of Figures

List of Tables

1 Introduction

As our devices keep getting packed with more features every generation, the complexity of the embedded control systems inside these devices is bound to increase. However, these devices are still required to be compact or maybe even smaller than their previous versions. This limits the resources like computational power and memory on the circuit boards at the core of the devices.

Traditionally, a common practice was to use low level languages like C to program embedded systems without the use of complete operating systems. However, the C programming language has a small overhead when it comes to memory usage and runtime. Developers have to manually manage memory allocations which might not be everyone's strongest suit, resulting in software bugs which could affect the safety and security of these devices. Moreover, C does not offer a good level of data abstraction and security when compared to higher level programming languages.

Rust is syntactically similar to C++ but can offer a high level of memory safety and management without garbage collection [1]. In this thesis, we will explore the possibility and limitations of using higher level programming languages like C++ and Rust on memory-constrained embedded systems by implementing an algorithm to generate sound of an engine based on the revolutions per minute(RPM) and also another algorithm in Quicksort so as to strengthen our conclusions.

1.1 Problem Statement

With the complexity of embedded systems increasing steadily, programming them with a low level language like C is cumbersome and time consuming. The main aims of the thesis are to:

- Analyze why higher-level languages have not been primarily used and pinpoint the benefits and risks of using higher level languages in embedded systems.
- Compare and contrast the performance of multiple algorithms by implementing them on two different high level languages, namely C++ and Rust.

1.2 Contribution

The thesis contributes the following:

- Design of an algorithm for artificial engine sound generation in C++ and Rust.
- Implementation of the Quicksort algorithm in C++ and Rust.

- An evaluation of the performance of these languages in terms of execution time, memory consumption and development time on a STM32 evaluation board.

1.3 Scope

Evaluation of the performance of C++ and Rust in an embedded device by implementing will be the scope of this master thesis.

Determining which programming language is superior is not the scope of this thesis. The performance evaluation of these languages is limited to a single hardware device and is not done on multiple hardware. The performance metrics considered for evaluation are execution time, memory consumption and development time only.

1.4 Method

The algorithms will first be implemented in MATLAB to understand the concept in order to efficiently implement them in C++ and Rust using all the advanced features that each language has to offer. The algorithms will be run on STM32 evaluation board with limited resources to estimate their execution time and memory consumption in real time.

The execution time of the algorithm on the evaluation board will be measured by getting system clock timer values at the beginning and the end of the algorithm execution by setting breakpoints in the IDE [2].

The IDE has an inbuilt memory analyzer tab which indicates how much stack memory has been used [2]. As for the heap memory, we will set the heap memory addresses to a known value (like 1) before execution by using STM32 ST-Link Utility software. After the execution, we will calculate the amount of heap memory used by checking the over-written addresses.

The development time will be measured for multiple implementations of the same algorithm, so as to give us more samples for strengthening our conclusions.

1.5 Thesis outline

The rest of the thesis is divided into the following chapters:

- Chapter 2 introduces the programming languages with their features and describes theory behind the working of an engine and the model of the engine chosen to write the algorithm.
- Chapter 3 presents the design choices as well as design and evaluation methodology.
- Chapter 4 describes the the results obtained after evaluation.

- Chapter 5 summarizes and concludes the thesis.

2
Technical Background

This chapter deals with the technical aspects of the thesis with a background of using high level languages in embedded systems and the main focus on the important features of each language used in our algorithm. It also explains the concepts behind sound generation and how the entire engine model is designed. Finally, we discuss about the hardware used and how Rust is run on the hardware.

2.1 High Level Languages in Embedded Systems

Real time and embedded systems are naturally reactive as their working is affected by external factors, for which the systems need to be resilient and robust. Hence, safety and reliability are really important factors, which are provided by high level languages as they allow higher level of data abstraction from machine language and efficient memory handling. But, the downside is the speed / execution time, as low level languages (C, traditional C++) are much faster than high level languages. Real time systems with hard deadlines and timing constraints would be affected in this case due to the increased execution time of high level languages. Hardware support is another factor to be considered while choosing programming languages, as low level and traditional languages have support in almost every hardware, whereas for high level languages it is very dependent on factors like hardware architecture, compiler etc. For example, in our case, the Rust compiler [1] depends on LLVM[1] for generation of machine code. So in terms of hardware targets for Rust, it is understood that the ARM architecture supports Rust.

Some high level languages like Haskell have a runtime system which contains 50k lines of C code, then a garbage collection mechanism, a scheduler and more. This might be difficult for a memory-constrained embedded system to handle. But, for our case, Rust has less to no runtime due to its zero-cost abstractions which is an advantage as it takes Rust closer to the efficiency of a low level language in a sense. As previously stated, high level languages provide safety and reliability, for example high level languages like C++ provide safety using its classes and structures for data privacy and abstraction. Some other benefits in theory for high level languages are that they are programmer friendly, as in they are easy to write and debug. They are also less error prone, as debugging errors is much simpler in high level languages.

[1]LLVM compiler infrastructure is a set of compiler and toolchain technologies which is used to develop a front end for any programming language and also the back end of any instruction set architecture[1].

2.2 Modern C++

C++ is a multi-paradigm programming language, as it supports both procedural and object oriented programming. It was created by Bjarne Stroustrup as an extension of the C programming language with classes [3]. The main idea behind the design of C++ was to build a systems programming language for embedded and resource constrained software for large systems with performance, efficiency and flexibility of use as its main features [4]. The latest versions of C++ from C++11, C++14 and later are classified as “Modern C++” [5], a term becoming prevalent nowadays. These versions rely on using the core features of the C++ language [5] like standard libraries, exceptions and templates, rather than just using C with classes. Traditional C++ focuses more on Object Oriented Programming, while Modern C++ uses different programming styles such as generic, object, procedural and modular [5]. In this thesis, we analyze how the real time performance is affected by exploiting some key aspects of Modern C++ discussed below.

2.2.1 Classes

This is one of the main features that attracts many developers even today. Classes hold different types of data as members or fields and code that can be used to access them are the methods or member functions. Objects are instances of such classes. Since data and the functions that can modify the data are encapsulated in classes, C++ provides a certain level of safety compared to C avoiding misuse of data. As shown is listing 1, class DelayLine has two data members, delay of type double and data which is a vector of doubles. These are declared in the private scope of the class so that only the member functions like push can access them.

```
class DelayLine
{
    private:
    //class members
    double delay{0.0};
    std::vector<double> data;

    public:
    //class methods
    void push(double &value, int &pos)
    {
        data[pos]=value;
    }
};
```

Listing 1: An Example Class

2.2.2 Tuples

A tuple is an object of fixed size which holds elements of same or different data types as shown in listing 2. They are part of the standard library. Tuples are immutable by default and have a finite structure i.e new elements cannot be added or deleted to/from a tuple. They can be constructed and deconstructed using standard library functions and can also be used to return multiple values from functions. In the example below, the function Tuple is used to construct and return a tuple which may then be split up and stored in different variables using the "tie" function.

```
std::tuple<char, int> Tuple() {
//constructing a tuple
return std::make_tuple('a', 1);
}
int main {
int x,y;
 //deconstructing a tuple
 std::tie(x,y)= Tuple();
 std::cout<<"The value of x is << x << "and y is " << y;
}

Output :
The value of x is a and y is 1.
```

Listing 2: A Tuple

2.2.3 Templates

Function templates are used to operate on generic types thus allowing us to use the same function independent of the data type of its parameters. Template is a type that has not been specified but it can be used by the function as any other data type. This is done by using template parameters as an argument to the function, as shown in listing 3. In the example below, the same function Max prints the maximum value between two values regardless of the data type.

```
//template parameter declaration
template<typename T>

//templatized function
T Max(T x, T y)
{
    return (x>y)?x:y;
}
int main()
{
    int x = Max(5,10);
    float y = Max(4.5, 6.8);
    char z = Max('a', 'b');
    std::cout<<x<<,<<y<<,<<z;
}

Output:
10, 6.8, b.
```

Listing 3: Template function

2.2.4 Auto

Much like other high level languages, the auto keyword in C++ lets the compiler deduce the type of data automatically while compiling, instead of having to explicitly define the data type. The auto keyword in C is just a storage class specifier, to indicate the variable to be local to a block [5].

2.2.5 Vectors

Vectors are similar to the concept of arrays in C but are dynamically allocated according to the user's needs. This has a big advantage over arrays due to the fact that arrays have to be deallocated explicitly if defined dynamically. Vectors are automatically deallocated from the heap as the destructor of vector is called when the method executes. C++ standard library provides lot of functions to access and modify vectors. Moreover, when arrays are passed to a function it is done by passing a pointer to the first element so we also have to send their size. But in the case of vectors, a copy of the vector is created and passed thus eliminating the need to send the size as well as shown in listing 4. Passing by reference is ideal for vectors of large sizes.

```
// The vector here is a copy of vector in main()
void function(std::vector<int> data)
{
   data.push_back(30);
}

int main()
{
    std::vector<int> t;
    t.push_back(10);
    t.push_back(20);

    function(t);

    // vector remains unchanged after function call
    for (int i=0; i<t.size(); i++)
       cout << t[i] << " ";

    return 0;
}
Output:

10 20
```

Listing 4: Vectors

2.3 Rust

Rust is also a multi paradigm language, introduced primarily for performance and safety. Rust is one of the recent additions to the family of systems programming languages that supports development of operating systems and software [6], which requires direct access to the hardware. Rust is a strongly statically typed language where the data types of the variables are available at compile time instead of run time which increases the performance. They are either specified by the user or inferred by the compiler thereby making error detection and bug removal easier at compile time.

The performance is further increased by the reduced runtime overhead due to absence of a garbage collector like C++. Rust has a type system and a borrow checker to manage memory by ensuring that references do not outlive the data they are referring to, thereby eliminating the concept of NULL (and wild) pointers and an entire class of bugs caused by unsafe memory management [1].

2.3.1 Mutability

In Rust, variables are declared using the `let` keyword and are immutable by default i.e, their values cannot be modified later. Any attempt to change the value of an immutable variable will generate a compile time error, as shown in listing 5. However, the `mut` keyword allows us to modify the variables. The variables can also be typed explicitly or determined by the compiler at compile time ensuring safety and less runtime overhead.

```rust
 fn main() {
    let x = 5;    //immutable variable
    println!("The value of x is: {}", x);
    x = 6;
    //compile time error
    println!("The value of x is: {}", x);
    let mut y=6;  //mutable variable
    println!("The value of x is: {}", y); //Output is 6.
    y=4;
    println!("The value of x is: {}", y); //Output is 4.
}
```

Listing 5: Variables and Mutability

2.3.2 Ownership

Ownership is one of the unique features of Rust that helps in safe memory management without a garbage collector. Every value in Rust has a variable called owner. There can be only one owner for a value and when the owner goes out of scope the value will be dropped (the lifetime of a variable is the point from which it was created till the end of the current scope). But unlike C and C++, as soon the owner is out of scope, the "drop" function is called by Rust which deallocates the memory. This is how Rust deals with memory management efficiently without a garbage collector.

When one variable is copied to another as shown in listing 6, Rust copies the details of the pointer to the memory instead of copying the data into the second. However, both variables will try to free the same memory when they go out of scope. This is called the double free error and freeing memory twice leads to memory corruption and this is where Rust's ownership comes into play. To ensure memory safety, Rust considers s1 to be no longer valid and therefore it doesn't free anything when s1 goes out of scope. s2 gets ownership of the resource from the first and only s2 alone can free the memory location when it goes out of scope. However, for primitive data types like `int`, Rust copies the data so the ownership does not change.

```
 fn main() {
    let s1 = String::from("hello");
    //ownership moved to s2
    let s2 =s1;
    //compilation error : value moved
    println!("{}", s1);
}
```

Listing 6: Ownership

2.3.3 Borrowing

In order to share variables past their scope without taking ownership, Rust uses a technique called *borrowing* which is basically using references to the owned variable instead of copying the original variable. This is how functions return values in Rust. The ownership of the value is temporarily borrowed when the function executes and the lifetime of this borrow is only within the function as the ownership is transferred back to the original variable when the function finishes executing. The scope of these borrows cannot exceed the scope of the owner and borrows are immutable by default. But Rust allows one mutable borrow per variable. So each resource has exactly one mutable borrow and/or an arbitrary number of immutable as shown in in listing 7. This avoids the race condition when two pointers access or modify the same data without synchronization.

```rust
fn main() {
    let mut s = String::from("hello");

    let r1 = &s; // no problem
    let r2 = &s; // no problem
    let r3 = &mut s;
    // Compile time error
    //mutable borrow occurs when r1 and r2 are still in scope
    println!("{}, {}, and {}", r1, r2, r3);

 }

fn main() {
    let mut s = String::from("hello");

    let r1 = &s; // no problem
    let r2 = &s; // no problem
    println!("{} and {}", r1, r2);
    // r1 and r2 are no longer used after this point

    let r3 = &mut s; // no problem
    println!("{}", r3);
}
```

Listing 7: Immutable Borrows

2.4 Artificial engine sound generation

The performance of the two languages Rust and C++ will be evaluated by applying them on resource intensive applications, so as to make a holistic comparison. In this case, the application chosen is artificial engine sound generation. The electric vehicles produce less engine sound, especially silent at lower speeds, and at higher speeds, the tire sounds become more dominant. Since the electric vehicles do not produce much sound, they are considered to be very environment friendly. But, one issue is that, these silent electric vehicles could cause some problems to the pedestrians, cyclists and visually impaired people. Hence, global governing bodies have been trying to impose laws on minimum level of sound for electric vehicles [7].

The generation of the engine sound is based on the paper [8]. The sound design imitates a combustion engine. The working of a four stroke engine is as follows: the crankshaft rotates such that the piston moves in different phases. There are four strokes namely, intake, compression, ignition and exhaust. Firstly, during the intake cycle, the fuel valve is opened allowing fuel/air to enter cylinder, while the exhaust is closed, until the piston reaches the bottom. Then, as the piston moves up, it compresses the mixture until a spark ignites the fuel mixture. So, when the piston moves past the top, an explosion occurs which gives the crankshaft a rotational energy. Hence, the four stroke cycle will continue as long as there is a supply of fuel and air[9].

2.5 Hardware

The testing of the different algorithms in Rust and C++ is done on an STM32H753I-EVAL2 board [10] because evaluation boards are generally used for industry level prototype testing. Figures 2.1 and 2.2 show the front and back of the STM32H753I-EVAL2 board. The significant reasons for the choice of this evaluation board are its compatibility with Rust and the high end range of specifications among all the evaluation boards such as high RAM, SAI Audio Support DAC and ADC as a sound source.

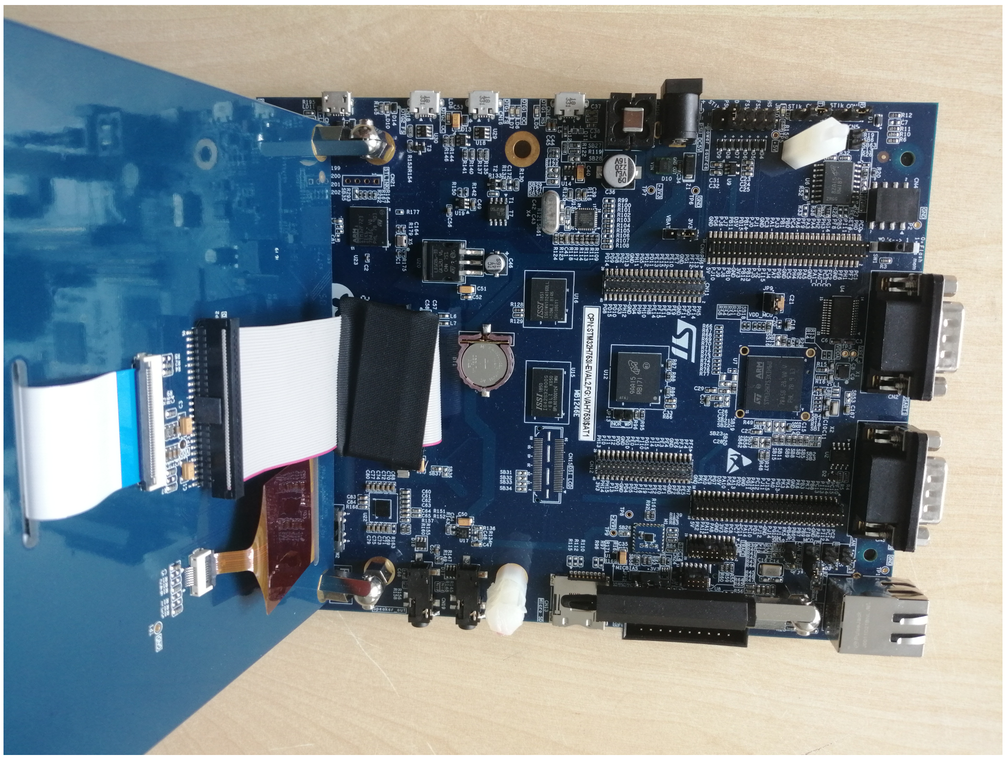


Figure 2.1: STM32H753I-EVAL2 evaluation board - Front without LCD

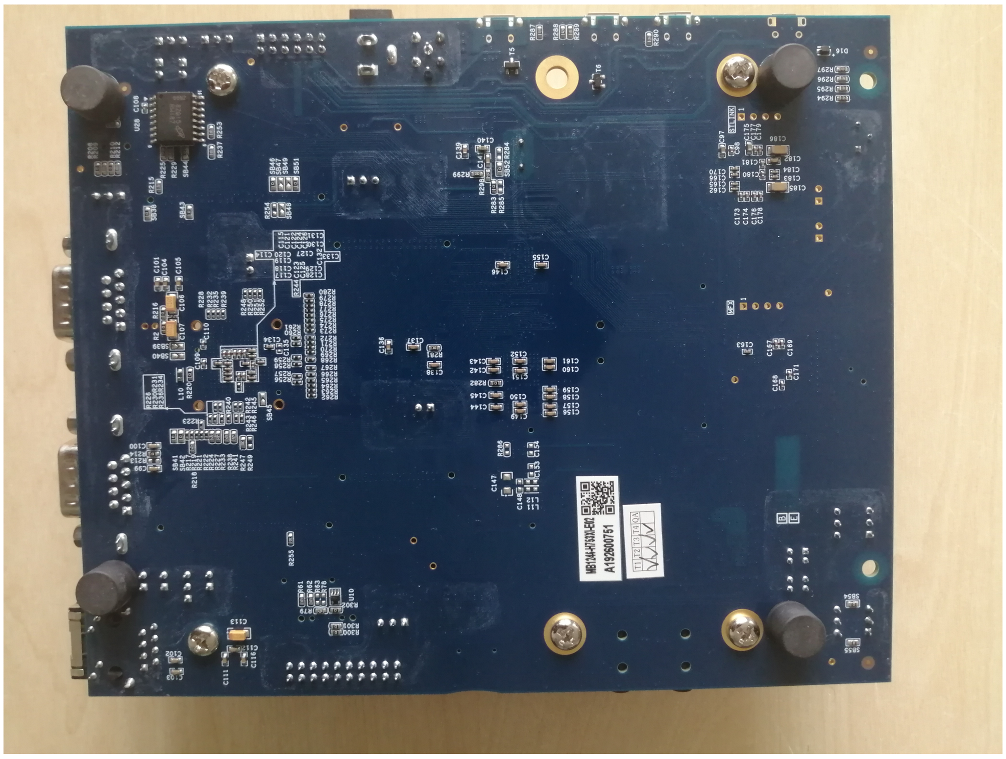


Figure 2.2: STM32H753I-EVAL2 evaluation board - Back

2.6 Rust for embedded devices

Rust in embedded devices is run in a bare metal environment by using the `no-std` attribute, where an attribute is a metadata applied to a module/crate. A crate is a compilation unit in Rust, in the form of a binary/library. The standard library `libstd` requires system integration as it provides a way of accessing OS abstractions and also provides a runtime for setting up stack overflow protection, processing command line arguments etc. `No-std` is a crate level attribute which specifies that the crate will link to the core-crate instead of the std-crate. The core-crate is a platform-agnostic subset of the `std` crate which provides APIs for primitives data types like floats, strings, slices etc as well as processor features like atomic operations and SIMD instructions. The `libcore` crate does not make any assumptions about the system that the program will run on.

Additionally, we will also use Rust's toolchain installer `rustup` to add targets for cross compilation support for the ARM Cortex-M architectures. The debugger we are going to use is GDB. But GDB [11] will not be able to communicate with ST-link debugging hardware on the board. So we are going to use OpenOCD [12] that translates between GDB's TCP/IP based remote debug protocol and ST-Link's USB based protocol. OpenOCD also helps with breakpoint/watchpoint manipulation, reading and writing to the CPU registers, continuing CPU execution after a debug event etc which will be useful when we measure execution time. STM32CubeIDE has a Hardware Abstraction Layer(HAL) embedded software that provides Application programming interface(API) to the device's peripherals. `Stm32h7` crate provides Rust support for all STM32 microcontrollers including board specific APIs and HAL.

3 Methods

This chapter deals with the methods used for the algorithms and performance analysis. Section 3.1 explains the concepts and methods used to design the entire engine model. Section 3.2 deals with the design of the Quicksort algorithm. Finally, section 3.3 discusses the methods used to measure each of the performance parameters.

3.1 Artificial Engine Sound Algorithm

3.1.1 Four Stroke Cycle

The starting step of the sound design algorithm is to mathematically represent the four strokes of the engine. This step leads to a consistency in the physical processing. The focus is held mainly on the acoustic contribution of these strokes. Each stroke is represented as follows as in equations 3.1, 3.2, 3.3 and 3.4, where x and t are crankshaft positions and ignition time respectively.

$$e(x) = \begin{cases} -\sin(4\pi x) & \text{if } 0.75 < x < 1 \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

$$i(x) = \begin{cases} \sin(4\pi x) & \text{if } 0 < x < 0.25 \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

$$s(x) = \begin{cases} \sin(2\pi(x - 0.5)/t) & \text{if } 0.5 < x < 0.5 + t/2 \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

$$p(x) = \cos(4\pi x) \tag{3.4}$$

The graph below in figure 3.1 depicts the MATLAB simulation results of the functions of the four stroke engine.

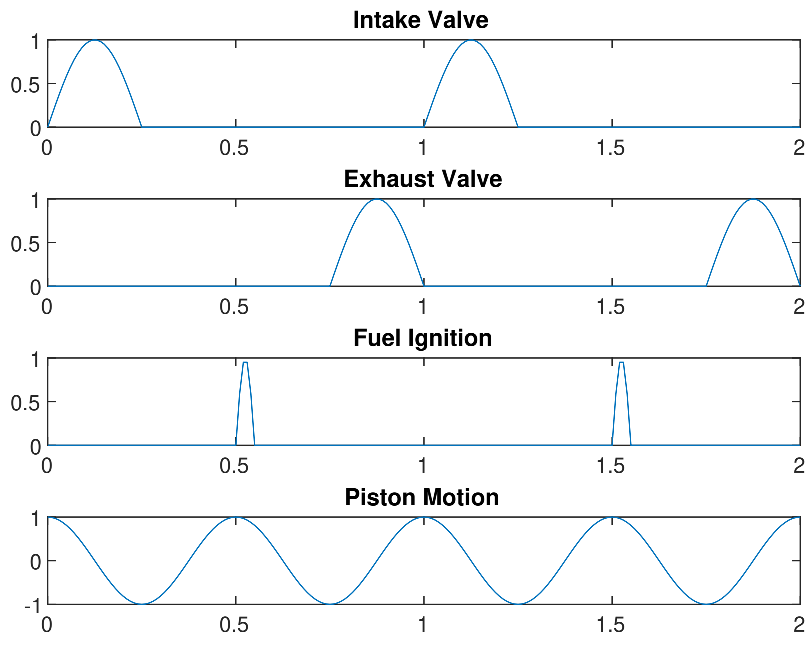


Figure 3.1: The valve functions

3.1.2 Digital Waveguides

An internal combustion engine can be considered as a set of interconnected pipes, when just the acoustics are the priority. Here, the resonant modes of the engine are excited by the aerodynamic interactions. Hence, in our design, these resonances are simulated using digital acoustic waveguides.

In general, a digital waveguide is constructed as shown in figure 3.2, consisting of two delay lines, where one's output is fed back as input of the other. The digital waveguide imitates the pipe where the acoustic wave moves from one end to the other and bounces back. As shown in figure 3.2, α and β are the valve modulation functions, as they are used to control the reflectivity of the side. Hence, the delayed wave produces an interference to the signal, thereby simulating the resonances in the tube[8].

The delay lines have been designed using multiple methods, such as using first in first out(FIFO) buffers and zero padded buffers. The FIFO buffers are of fixed length and push out data every time the buffer is full and the zero padded buffers were designed in such a way that, the incoming signal is delayed for a certain number of samples by padding zeros to the signal, depending on the length of the pipes using equation 3.5, which simulates the time taken for the wave to reach from one end to the other end of the pipe. Then, the signals are factored by a coefficient less than 1 to imitate the energy re-circulation. Equation 3.5 is used for delay samples ($t_{samples}$) calculation, where D is the distance, S is the sample rate and c is the speed of sound, which is 343 m/s.

$$t_{samples} = \frac{D \cdot S}{c} \tag{3.5}$$

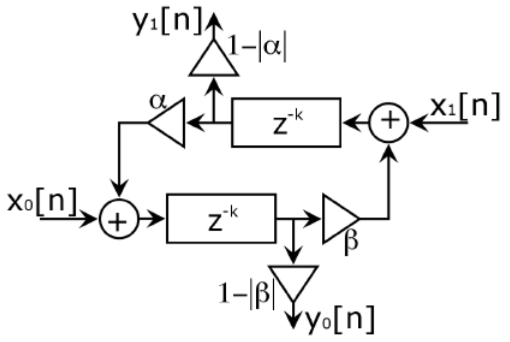


Figure 3.2: A Digital Waveguide

3.1.3 Engine Model

The engine sound model is designed as shown in figure 3.3. The system overall has five inputs : the intake valve, piston motion, fuel ignition, exhaust valve, and a filtered white Gaussian noise. The low pass filtered white noise is added to the intake side, in order to simulate the air-fuel turbulence in the combustion chamber. At the other end, there are three outputs/sound components such as intake, engine vibration and the outlet.

The process of sound generation starts in the cylinder. The engine vibration sound is produced in the cylinder by adding the piston motion and fuel ignition signals, and later low pass filtering the same. Meanwhile, the cylinder sound is then shared to both the intake and the exhaust sides. So, the intake and the exhaust valves are primarily used to modulate the feedback coefficients of the waveguide. Hence, the intake sound is finally obtained by adding the low pass filtered white noise with the intake valve modulated cylinder sound. The exhaust side consists of a straight pipe, muffler and an outlet. Here, the cylinder sound is modulated by the exhaust valve and sent through the straight pipe, which is designed as a waveguide. Then, it is passed through the muffler, which is used to reduce the exhaust noise. Hence, it is designed as a group of four delay elements to create an interference to the signal thereby reducing the noise. Thus, the exhaust sound is collected at the outlet, thereby completing the trio of significant sound components. Theoretically, in case of a four cylinder engine, the cycles of each cylinder is shifted equally in phase, to distribute the power uniformly to the crankshaft. In reality, the power is not necessarily uniformly distributed, since the engine revolutions are not very precise, so in design an offset fluctuation to the phase is considered.

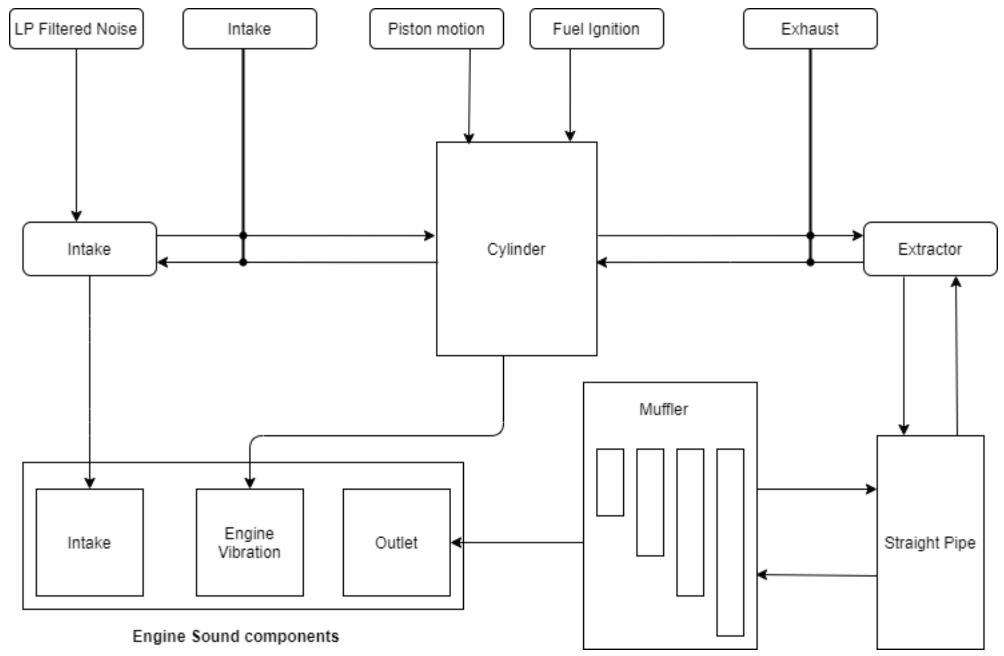


Figure 3.3: Design of the engine model

3.2 Quicksort Algorithm

Performance analysis of the languages should not just be restricted to one application, since the sample size is too small to generalize the findings. Hence, to reaffirm the same, the performance analysis is also done for the Quicksort algorithm of a large sized array. Quicksort algorithm is a divide and conquer approach[13], where the large array is continually subdivided into smaller groups which are in turn compared and sorted.

This algorithm was implemented in C++ using the qsort() function, which is part of the C++ standard library. The syntax for the function and its usage are shown in listing 8, and it requires an extra comparison function to compare within the sub-arrays for sorting purposes. Similarly in Rust, a crate called Quicksort is used to implement the same algorithm, and it does not require any extra functions.

The Quicksort algorithm was designed in two methods for each language, one using a pre-defined function, as in a function from the standard library in case of C++ or a crate in case of Rust. The other method was to write the algorithm without using a pre-defined function and rather implement it on our own. The difference in implementations of the pre-defined Quicksort functions in Rust and C++ created an uncertainty during the performance evaluation, hence the idea of implementing the algorithm in a similar manner for both the languages was put forth, for a fair comparison.

As explained above, the implementation has two important functions namely, the partition function and the swap function. A pivot element is chosen to partition the entire array into two. The partition function is where the array is partitioned in a way that the numbers less than pivot are on one side of the pivot and greater numbers on the other side, and is done so on till the sub arrays are smaller and sorted.

```
qsort(arr,num,sizeof(int),compare);
int compare(const void* a, const void* b);
```

Listing 8: qsort() function prototype

3.3 Performance Analysis Methods

The three parameters we will be measuring are execution time, memory usage and development time.

3.3.1 Execution Time Analysis

For measuring execution time on the board while running the C++ program, we use the debug mode in STM32CubeIDE which allows us to set breakpoints. We record the time stamps of the system clock on the board by using the command *HAL_GetTick()* (listing 9) which returns the time in milliseconds elapsed since startup. The clock configuration is set at 100 MHz, as it is the highest possible hardware configuration for the execution of test program.

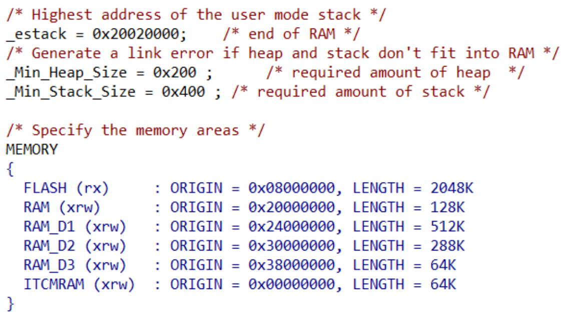
While running Rust, we use OpenOCD to set the breakpoints. A timer is set up with a frequency of 1kHz and we record the counter value of the timer using the *timer.counter()* function (provided by the HAL in the stm32h7 crate) at the start and end of execution.

```
int main(void)
{
  /* USER CODE BEGIN 1 */

  uint32_t startx=0, endx=0,timer_valuex=0;
  /* USER CODE END 1 */

  /* MCU Configuration---------------*/

  /* Reset of all peripherals,
  Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */
  memset(heap_start_address, '%', 16384);
  memset(stack_start_address, '%', 1536);
  memset(0x20000000, '$', 128*1024);
  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_TIM6_Init();
  /* USER CODE BEGIN 2 */
  // HAL_UART_MspInit(&huart1);
  HAL_TIM_Base_Start_IT(&htim6);
  startx=HAL_GetTick();
  run_engine();
  endx=HAL_GetTick();
  timer_valuex=endx -startx;
  HAL_TIM_Base_Stop_IT(&htim6);
```

Listing 9: Execution time measurement in STM32CubeIDE

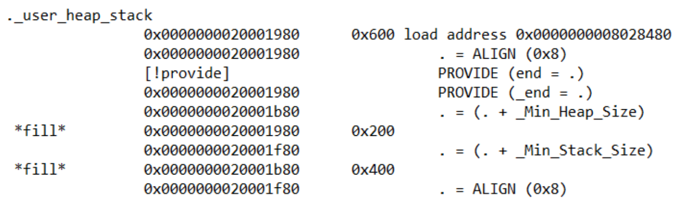
3.3.2 Memory Usage

The memory on the STM32H7 board is mapped into several regions(see figure 3.4). Using the IDE, we set the minimum amount of heap and stack memory to 0x200 and 0x400 bytes as shown in figure 3.5. In order to measure the memory used by the program, we use the STM32 ST-Link utility software which gives the user direct access to read and write the onboard memory. The memory is set to hold '$' using the command *memset* before running the program as shown in listing 9 and the amount of memory used(in bytes) is measured by subtracting the start and end addresses rewritten after the execution of the program in the stack and heap memory regions as depicted in the figures 3.6, 3.7, 3.8 and 3.9.



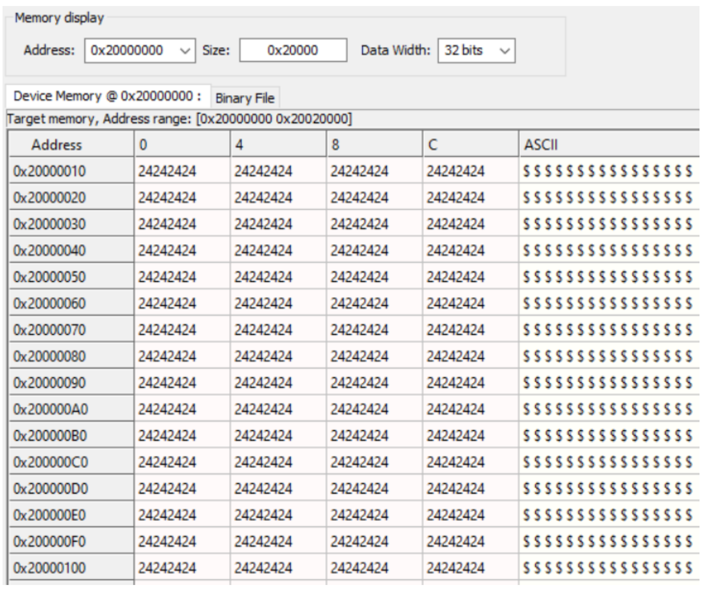
```
/* Highest address of the user mode stack */
_estack = 0x20020000;    /* end of RAM */
/* Generate a link error if heap and stack don't fit into RAM */
_Min_Heap_Size = 0x200 ;      /* required amount of heap  */
_Min_Stack_Size = 0x400 ; /* required amount of stack */

/* Specify the memory areas */
MEMORY
{
  FLASH (rx)      : ORIGIN = 0x08000000, LENGTH = 2048K
  RAM (xrw)       : ORIGIN = 0x20000000, LENGTH = 128K
  RAM_D1 (xrw)    : ORIGIN = 0x24000000, LENGTH = 512K
  RAM_D2 (xrw)    : ORIGIN = 0x30000000, LENGTH = 288K
  RAM_D3 (xrw)    : ORIGIN = 0x38000000, LENGTH = 64K
  ITCMRAM (xrw)   : ORIGIN = 0x00000000, LENGTH = 64K
}
```

Figure 3.4: Memory Mapping for STM32H7



```
._user_heap_stack
                0x0000000020001980      0x600 load address 0x0000000008028480
                0x0000000020001980                . = ALIGN (0x8)
                [!provide]                        PROVIDE (end = .)
                0x0000000020001980                PROVIDE (_end = .)
                0x0000000020001b80                . = (. + _Min_Heap_Size)
 *fill*         0x0000000020001980      0x200
                0x0000000020001f80                . = (. + _Min_Stack_Size)
 *fill*         0x0000000020001b80      0x400
                0x0000000020001f80                . = ALIGN (0x8)
```

Figure 3.5: Heap Stack Memory Map



Memory display

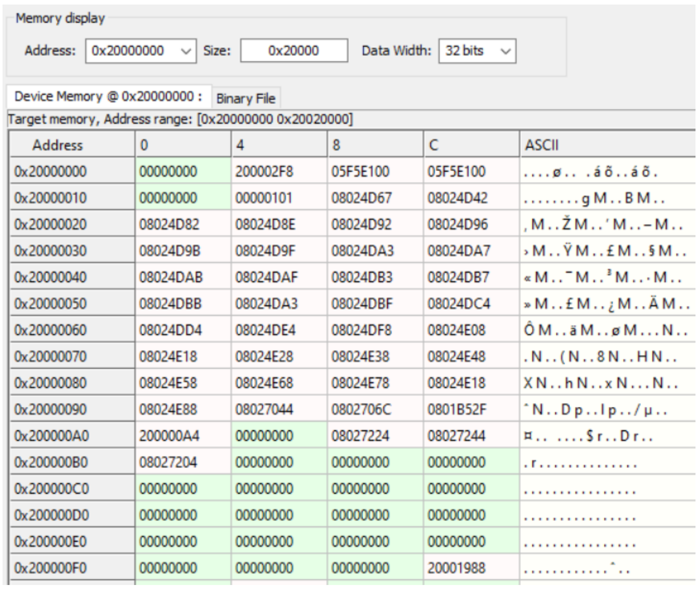
Address: 0x20000000 Size: 0x20000 Data Width: 32 bits

Device Memory @ 0x20000000 : Binary File

Target memory, Address range: [0x20000000 0x20020000]

Address	0	4	8	C	ASCII
0x20000010	24242424	24242424	24242424	24242424	$$$$$$$$$$$$$$$$
0x20000020	24242424	24242424	24242424	24242424	$$$$$$$$$$$$$$$$
0x20000030	24242424	24242424	24242424	24242424	$$$$$$$$$$$$$$$$
0x20000040	24242424	24242424	24242424	24242424	$$$$$$$$$$$$$$$$
0x20000050	24242424	24242424	24242424	24242424	$$$$$$$$$$$$$$$$
0x20000060	24242424	24242424	24242424	24242424	$$$$$$$$$$$$$$$$
0x20000070	24242424	24242424	24242424	24242424	$$$$$$$$$$$$$$$$
0x20000080	24242424	24242424	24242424	24242424	$$$$$$$$$$$$$$$$
0x20000090	24242424	24242424	24242424	24242424	$$$$$$$$$$$$$$$$
0x200000A0	24242424	24242424	24242424	24242424	$$$$$$$$$$$$$$$$
0x200000B0	24242424	24242424	24242424	24242424	$$$$$$$$$$$$$$$$
0x200000C0	24242424	24242424	24242424	24242424	$$$$$$$$$$$$$$$$
0x200000D0	24242424	24242424	24242424	24242424	$$$$$$$$$$$$$$$$
0x200000E0	24242424	24242424	24242424	24242424	$$$$$$$$$$$$$$$$
0x200000F0	24242424	24242424	24242424	24242424	$$$$$$$$$$$$$$$$
0x20000100	24242424	24242424	24242424	24242424	$$$$$$$$$$$$$$$$

Figure 3.6: Heap Memory after erase



Memory display

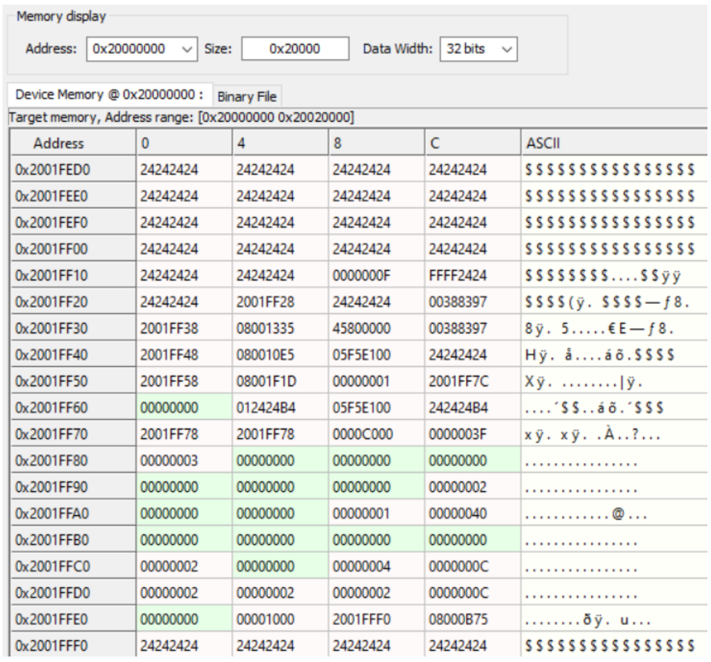
Address: 0x20000000 Size: 0x20000 Data Width: 32 bits

Device Memory @ 0x20000000 : Binary File

Target memory, Address range: [0x20000000 0x20020000]

```
Address     0         4         8         C         ASCII
0x20000000  00000000  200002F8  05F5E100  05F5E100  ....ø.. .áõ..áõ.
0x20000010  00000000  00000101  08024D67  08024D42  ........gM..BM..
0x20000020  08024D82  08024D8E  08024D92  08024D96  ‚M..ŽM..’M..–M..
0x20000030  08024D9B  08024D9F  08024DA3  08024DA7  ›M..ŸM..£M..§M..
0x20000040  08024DAB  08024DAF  08024DB3  08024DB7  «M..¯M..³M..·M..
0x20000050  08024DBB  08024DA3  08024DBF  08024DC4  »M..£M..¿M..ÄM..
0x20000060  08024DD4  08024DE4  08024DF8  08024E08  ÔM..äM..øM...N..
0x20000070  08024E18  08024E28  08024E38  08024E48  .N..(N..8N..HN..
0x20000080  08024E58  08024E68  08024E78  08024E18  XN..hN..xN...N..
0x20000090  08024E88  08027044  0802706C  0801B52F  ˆN..Dp..lp../µ..
0x200000A0  200000A4  00000000  08027224  08027244  ¤.. ....$r..Dr..
0x200000B0  08027204  00000000  00000000  00000000  .r..............
0x200000C0  00000000  00000000  00000000  00000000  ................
0x200000D0  00000000  00000000  00000000  00000000  ................
0x200000E0  00000000  00000000  00000000  00000000  ................
0x200000F0  00000000  00000000  00000000  20001988  ............ˆ..
```

Figure 3.7: Heap Memory after write



Memory display

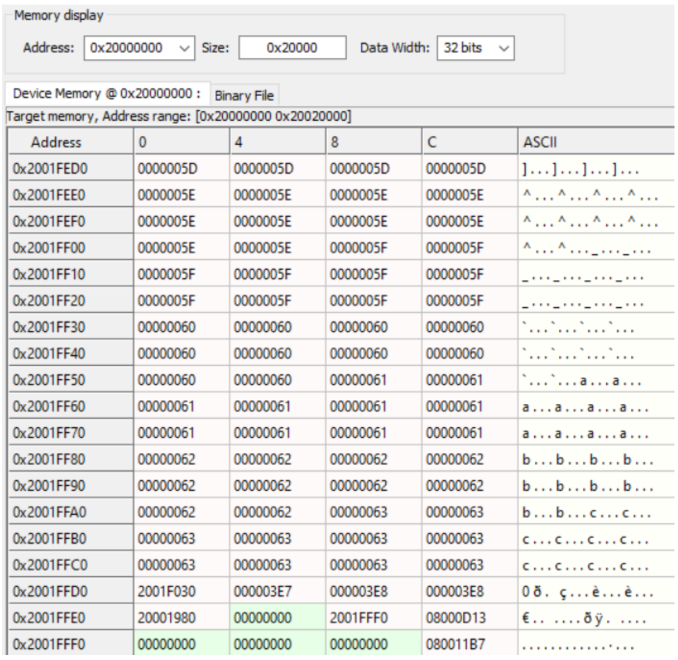
Address: 0x20000000 Size: 0x20000 Data Width: 32 bits

Device Memory @ 0x20000000 : Binary File

Target memory, Address range: [0x20000000 0x20020000]

```
Address     0         4         8         C         ASCII
0x2001FED0  24242424  24242424  24242424  24242424  $$$$$$$$$$$$$$$$
0x2001FEE0  24242424  24242424  24242424  24242424  $$$$$$$$$$$$$$$$
0x2001FEF0  24242424  24242424  24242424  24242424  $$$$$$$$$$$$$$$$
0x2001FF00  24242424  24242424  24242424  24242424  $$$$$$$$$$$$$$$$
0x2001FF10  24242424  24242424  0000000F  FFFF2424  $$$$$$$$....$$ÿÿ
0x2001FF20  24242424  2001FF28  24242424  00388397  $$$$(ÿ. $$$$—ƒ8.
0x2001FF30  2001FF38  08001335  45800000  00388397  8ÿ. 5.....€E—ƒ8.
0x2001FF40  2001FF48  080010E5  05F5E100  24242424  Hÿ. å....áõ.$$$$
0x2001FF50  2001FF58  08001F1D  00000001  2001FF7C  Xÿ. ........|ÿ.
0x2001FF60  00000000  012424B4  05F5E100  242424B4  ....´$$..áõ.´$$$
0x2001FF70  2001FF78  2001FF78  0000C000  0000003F  xÿ. xÿ. .À..?...
0x2001FF80  00000003  00000000  00000000  00000000  ................
0x2001FF90  00000000  00000000  00000000  00000002  ................
0x2001FFA0  00000000  00000000  00000001  00000040  ............@...
0x2001FFB0  00000000  00000000  00000000  00000000  ................
0x2001FFC0  00000002  00000000  00000004  0000000C  ................
0x2001FFD0  00000002  00000002  00000002  0000000C  ................
0x2001FFE0  00000000  00001000  2001FFF0  08000B75  ........ðÿ. u...
0x2001FFF0  24242424  24242424  24242424  24242424  $$$$$$$$$$$$$$$$
```

Figure 3.8: Stack Memory after erase



Memory display

Address: 0x20000000 Size: 0x20000 Data Width: 32 bits

Device Memory @ 0x20000000 : Binary File

Target memory, Address range: [0x20000000 0x20020000]

Address	0	4	8	C	ASCII
0x2001FED0	0000005D	0000005D	0000005D	0000005D	]...]...]...]...
0x2001FEE0	0000005E	0000005E	0000005E	0000005E	^...^...^...^...
0x2001FEF0	0000005E	0000005E	0000005E	0000005E	^...^...^...^...
0x2001FF00	0000005E	0000005E	0000005F	0000005F	^...^..._..._...
0x2001FF10	0000005F	0000005F	0000005F	0000005F	_..._..._..._...
0x2001FF20	0000005F	0000005F	0000005F	0000005F	_..._..._..._...
0x2001FF30	00000060	00000060	00000060	00000060	\`...\`...\`...\`...
0x2001FF40	00000060	00000060	00000060	00000060	\`...\`...\`...\`...
0x2001FF50	00000060	00000060	00000061	00000061	\`...\`...a...a...
0x2001FF60	00000061	00000061	00000061	00000061	a...a...a...a...
0x2001FF70	00000061	00000061	00000061	00000061	a...a...a...a...
0x2001FF80	00000062	00000062	00000062	00000062	b...b...b...b...
0x2001FF90	00000062	00000062	00000062	00000062	b...b...b...b...
0x2001FFA0	00000062	00000062	00000063	00000063	b...b...c...c...
0x2001FFB0	00000063	00000063	00000063	00000063	c...c...c...c...
0x2001FFC0	00000063	00000063	00000063	00000063	c...c...c...c...
0x2001FFD0	2001F030	000003E7	000003E8	000003E8	0ð. ç...è...è...
0x2001FFE0	20001980	00000000	2001FFF0	08000D13	€.. ....ðÿ. ....
0x2001FFF0	00000000	00000000	00000000	080011B7	.............·...

Figure 3.9: Stack Memory after write

3.3.3 Development time

We consider the time from the beginning of writing the code till successfully running the program on the board to be the development time for that language. So we measure the number of days it took to write and successfully run both the algorithms on the board. The development time recorded here is strictly with respect to us and there is no proper way to generalize this time as it will be different for different software developers.

4 Results and Discussion

This chapter presents and discusses the results obtained from all the measurements and tests. We discuss and compare C++ and Rust with respect to execution time, memory consumption and development time for embedded devices.

4.1 Execution Time Analysis

The execution time analysis was done as explained in the previous section for the two algorithms in Rust and C++. The results for the Quicksort algorithm are shown in table 4.1. It is very clear that the execution time is significantly less for C++ than Rust, in both the cases, with or without pre-defined function. This is also the case in artificial engine sound algorithm to a certain extent, as shown in Table 4.2.

Table 4.1: Execution time analysis - Quicksort Algorithm for 1000 numbers ranging from 0 to 99

Execution Time	**User-defined function ms**	**Standard Library Function ms**
C++	29	6
Rust	732	494

Table 4.2: Execution Time analysis - Artificial Engine Sound Algorithm for RPM=8000 and Samples=7200

Execution Time	**Zero Padded Buffer s**	**Loop Buffer s**
C++	13.574	5.332
Rust	13.289	13.935

4.1.1 Discussion

The execution time is better for C++ in comparison with Rust, which was expected. But there are differences in the execution of these languages on the board as the standard library was used in the case of C++ and not in case of Rust, due to the incompatibility to run std library on bare metal environment. This leads to an addition of external crates to the Rust program which would subsequently increase the execution time. The external crates are used from the Rust community website developed by users, so they are not necessarily the most optimized versions of code. Further, the core library in Rust is minimal: it isn't even aware of heap allocation, nor does it provide concurrency or I/O [1]. In the case of user-defined Quicksort function, despite the usage of no external crates, the performance of Rust was significantly worse than C++. One reason might be that the compiler optimizations of LLVM compiler on ARM hardware is comparatively lesser than the GCC compiler [14].

Table 4.3: Execution Time analysis - Quicksort Algorithm for different array sizes

Execution Time	**30000 unsigned 32 Numbers s**	**60000 unsigned 16 Numbers s**
C++	0.497	1.397
Rust	1.989	3.193

The crates used for random noise generation alone cost close to 40% of the execution time, which brings the execution time of rest of the program in Rust at least close to C++. This also suggests that the features of Modern C++ were much more in effect in the engine sound algorithm as compared to the Quicksort algorithm, as only the standard library function was required in the Quicksort algorithm, which meant it was closer to traditional C++ than Modern C++, and it in turn leads to a low execution time. Moreover, the performance of Rust improved in comparison to C++ as the size of the array was increased, as shown in Table 4.3. Due to memory limitations in the board, the maximum array size possible was 60000, for which the ratio of execution time of Rust to C++ was 2, compared to 25 for 10000 numbers. From the observations, despite it still not being highly optimal, it can be said that Rust approaches the execution time of C++, as the resource usage increases.

4.2 Memory Analysis

The memory consumed after running each algorithm is shown below in table 4.4. And as expected Rust has performed better at handling memory compared to C++. The memory used is almost halved for both of our algorithms because of Rust's core principles about lifetime of variables and ownership, explained further in the discussion.

As for the Quicksort algorithm, the user defined algorithm uses much less heap memory compared to the C++ version. And we were also unable to flash the program on the board as Quicksort crate had issues with no-std enviroment.

Table 4.4: Memory Analysis - Artificial Engine Sound Algorithm for RPM=8000 and Samples=7200

Memory Consumption (in bytes)	**Zero Padded Buffer**		**Loop Buffer**	
	Heap Memory	**Stack Memory**	**Heap Memory**	**Stack Memory**
C++	8464	1680	7520	1344
Rust	5760	848	3632	864

Table 4.5: Memory Analysis - Quicksort Algorithm for 1000 numbers ranging from 0 to 99

Memory Consumption (in bytes)	**User defined function**		**Standard Library Function**	
	Heap Memory	**Stack Memory**	**Heap Memory**	**Stack Memory**
C++	80	7568	80	4336
Rust	48	7536	-	-

4.2.1 Discussion

As expected, Rust's type system and rules on lifetime of variables has improved efficient memory utilization. This is very important in embedded systems where there is limited on-board memory. Rust's ownership model where each value has a single owner at any given point of time and its rules of borrowing and references where the ownership of the value is transferred to a function are very different from the existing programming languages. They seem to have a significant impact by reducing the memory consumed by almost half in the case of engine sound generation algorithm. And as for the Quicksort algorithm, the amount of stack memory consumed is almost the same for both languages. This could be due to the fact that there are 1000 32 bit integers stored in an array in both cases. Therefore, based on our analysis and observations, we can see that Rust is more efficient in handling memory than C++ and it also provides memory safety guarantees which are important in embedded systems.

4.3 Development Time Analysis

Development time was a lot harder to measure and generalize than expected. We implemented the algorithm first in MATLAB in order to understand the working principle as described in [8]. Since this took a week longer than expected, the development time in C++ , i.e. the time it took from starting to write the algorithm in C++ till we successfully executed the program on the board was 3 weeks. This includes the time required to become familiar with Stm32CubeIDE, using the debug mode etc.

The Rust implementation of the same algorithm took each of us one week to write. Since we were writing the algorithm for the third time, we believe that is the reason for the programming time in Rust being much lower than C++ even though Rust was relatively new to both of us. However, getting the algorithm to run on the board was much harder than expected because of the need to find crates which run on the *no_std* environment. We also needed to learn to set breakpoints using OpenOCD, set up timers and clocks and extracting the values by programming which took some more time since STM32CubeIDE does not have Rust support. So the total development time in Rust was 4 weeks.

4.3.1 Discussion

Rust is a relatively new programming language compared to C and C++. With new concepts like the type system, mutability of variables, ownership and borrowing Rust could be overwhelming for beginners. However, Rust offers great documentation in the form of books on the Rust programming language website [1] and help from the Rust community provides a better learning experience to novice Rust programmers. Compile time checks and Rust's memory safety model helps reduce many classes of bugs at compile time. We believe that the compile time checks and helpful error messages from the compiler played a huge role in reducing the writing time of the code even though we did not use unsafe Rust or thread concurrency in our algorithms, where unsafe Rust feature allows the use of certain operations such as: dereferencing raw pointers, accessing fields of unions, calling unsafe functions, etc. The complexity of Rust increases as we begin to run Rust for embedded devices. Even though Rust includes proper documentation of the steps involved in cross compiling and successfully running on an embedded devices, it was challenging to find crates that worked in a no-std environment. Crates for filters and random number generation which were essential to our program had to be changed and the new crates for the no-std environment were written by users in the community and affected the execution time of the algorithm. Also data types like vectors which were available in the standard library could not be used directly.

As far as coding in C++ is concerned, we argue that the time it took for us to write code was more due to understanding of the algorithm itself rather than any issues with the language Rust and applying its features. The algorithm written in MATLAB was purely for understanding purposes so it did not have any data structures or efficient coding techniques. The entire structure of the algorithm changed for C++ when we used classes and templates for designing the engine. Also, once the algorithm was written, it was much easier to run it on the board as the IDE had support for C++ including all the standard libraries. Overall, we believe that if Rust continues to gain popularity and with more support from manufacturers and developers it is possible for Rust to have the same development time as C++.

5
Conclusion

With the objective to make a comparison study of high level languages in embedded memory constrained systems, the thesis has met those requirements. From the observations made, we can conclude that Rust has a better memory management, with a caveat of higher execution time, while also improving safety and reliability. But, as the number of testing algorithms/benchmarks is small, the strength of the conclusions may not be high. Moreover, only one hardware was used for all the tests, which may also weaken the argument. Despite these arguments, we tested different scenarios to validate our results. For example, one such observation in the Rust vs C++ execution time comparison was that, when the size of the array increases Rust's performance improves in comparison with C++. Regarding development time, Rust code took lesser time to write due to it being implemented last, but much more difficult to implement on the STM32H7 board that we used, as there was less support for the hardware and more to investigate.

With respect to the future scope of this topic area, the performance evaluation can be done on different hardware and architectures to reach a more stable conclusion. Further, on a general level, some organizations have started using Rust in their implementations. For example, Google started using Rust for low level implementations in their Android Open source project [15] due to its modern safety guarantees and good performance metrics, which indicates that high level languages like Rust could be used in embedded systems despite their drawbacks.

Bibliography

[1] The Rust Foundation, “The Rust programming language,” https://www.Rust-lang.org.

[2] “STM32CubeIDE - Integrated development environment for STM32,” https://www.st.com/en/development-tools/stm32cubeide.html.

[3] B. Stroustrup, *A History of C++: 1979–1991*. New York, NY, USA: Association for Computing Machinery, 1996, p. 699–769. [Online]. Available: https://doi.org/10.1145/234286.1057836

[4] B. Strostrup, *The C++ Programming Language (2nd Ed.)*. USA: Addison-Wesley Longman Publishing Co., Inc., 1991.

[5] “Modern c++ programming language,” https://docs.microsoft.com/en-us/cpp/cpp/welcome-back-to-cpp-modern-cpp?view=msvc-160.

[6] N. Adolfsson and F. Nilsson, “A Rust-based runtime for the internet of things,” Master’s thesis, Chalmers University of Technology, Gothenburg, Sweden, 2017. [Online]. Available: https://hdl.handle.net/20.500.12380/250074

[7] D. Washington, “Minimum sound requirements for hybrid and electric vehicles: Final environmental assessment(document submitted to docket number nhtsa-2011-0100. report no. dot hs 812 347),” *National Highway Traffic Safety Administration.*, 11 2016.

[8] S. Baldan, H. Lachambre, S. D. Monache, and P. Boussard, “Physically informed car engine sound synthesis for virtual and augmented environments,” in *2015 IEEE 2nd VR Workshop on Sonic Interactions for Virtual Environments (SIVE)*, 2015, pp. 1–6.

[9] A. Farnell, “Designing sound.” Cambridge, Massachusetts, USA: MIT Press, 2008.

[10] “STM32H753I Evaluation board databrief,” https://www.st.com/resource/en/data_brief/stm32h753i-eval.pdf, 2019.

[11] GDB Developers, “GDB: The GNU Project Debugger,” https://www.gnu.org/software/gdb, Copyright Free Software Foundation, Inc., 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.

[12] D. Rath, “Open On-Chip Debugger,” https://openocd.org.

[13] C. A. R. Hoare, "Algorithm 64: Quicksort," *Commun. ACM*, vol. 4, no. 7, p. 321, jul 1961. [Online]. Available: https://doi.org/10.1145/366622.366644

[14] J.-J. Kim, S.-Y. Lee, S.-M. Moon, and S. Kim, "Comparison of LLVM and GCC on the ARM platform," in *2010 5th International Conference on Embedded and Multimedia Computing*, 2010, pp. 1–6.

[15] Android open source project, "Android Rust Introduction," https://source.android.com/setup/build/rust/building-rust-modules/overview, 2022.