

Enhancing Error Correction Performance of Reed-Solomon Codes for Error and Erasure Environment on an STM32 Nucleo Board

Master Project Thesis

submitted by
Aditi Prasad
born in Kerala, India

Smart Sensors
Technische Universität Hamburg

March - August 2023

1. Reviewer : Prof. Dr.-Ing. Ulf Kulau
2. Reviewer : TBC

Affidavit

I hereby declare in lieu of an oath that I have produced this Master Project Thesis with the title of title „Enhancing Error Correction Performance of Reed-Solomon Codes for Error and Erasure Environment on an STM32 Nucleo Board” independently and without unauthorized outside help. I have not used I have not used any sources or aids other than those indicated, nor have I marked any quotations, either verbatim or in spirit. The work has not been submitted in the same or a similar form to any been submitted in the same or a similar form to any examination authority.

Hamburg, September 6, 2023
Place, Date



Aditi Prasad

Abstract

In the project, an extensive Reed Solomon coding is tested out for corrupted communication messages. A flexible model of Reed-Solomon code in a errors and erasure environment is proposed by stating requirements based on the implementation with performance measurement. The model was validated by experimental values. Findings indicate that the proposed model satisfies the primary attributes of a proper reed Solomon coding for a wide range of capacity, imperceptibility, and robustness. The research provides a standing base for further research implementation in a wide range of applications of Reed-Solomon codes.

Contents

Declaration	iii
List of Figures	vii
List of Tables	ix
Acronyms	xi
1 Introduction	3
1.1 Problem Description	3
1.2 Research Objectives	3
1.3 Research Design	4
1.3.1 STM32 Nucleo Board Description:	4
1.3.2 Hardware setup	4
1.4 Scope	5
2 Coding Theory	7
2.1 Hamming Distance	7
2.2 Types of Codes	7
2.3 Errors	7
2.4 Galois Fields	7
2.5 Arithmetic of Finite Field $GF(2^2)$	8
2.5.1 Field Elements	8
2.5.2 Addition Table	8
2.5.3 Multiplication Table	8
2.5.4 Reduced Forms	8
2.6 Construction of Finite Field $GF(2^3)[1]$	9
3 Reed-Solomon Coding	11
3.1 Characteristics of Reed-Solomon codes	11
3.1.1 Significance of Reed-Solomon Coding:	11
3.2 Methodology:	12
3.2.1 Encoding Reed-Solomon Codes	12
3.2.2 Decoding Reed-Solomon Codes	13
4 Implementation	17
4.1 ARM CORTEX M0+ Microcontroller	17
4.1.1 Profiling Limitations:	18
4.2 STM32CUBEIDE:	18

4.2.1	Configure STM32CubeIDE:	18
4.2.2	System Debug and Serial Wire Debug (SWD) Configuration	18
4.2.3	LPUART Configuration	21
4.2.4	TIM2 Configuration	21
4.2.5	Clock Configuration (HSI16)	22
4.2.6	Project	22
4.3	Functions	24
4.3.1	Check Multiple Inputs	24
4.3.2	DECLARATION(int mm, int nn)	24
4.3.3	ENCODE_RS()	25
4.3.4	DECODE_RS()	26
4.3.5	Profiling	27
4.4	Print Results:	29
5	Result and Analysis	31
6	Conclusion	35
A	Appendix	xiii
	Bibliography	xvi

List of Figures

3.1	Reed Solomon Codeword	11
3.2	(20,10,5) RS: encoder and decoder system	11
3.3	Reed Solomon Berlekamp Decoder system	14
4.1	STM32 L073RZT6	17
4.2	Start new project on STM32CubeIDE 1.12.1	19
4.3	Select Board and Create C project	20
4.4	System Configuration with debug serial wire	20
4.5	(a) & (b) LPUART Configuration for Pin 2 and Pin 3	21
4.6	Clock setting	23
4.7	Configure main.c	23
4.8	Open serial console of STM32CubeIDE	29
4.9	Open serial console of STM32CubeIDE	30
4.10	Console connection	30
5.1	(10,4,6) RS: Detect and Correct at Least 3 Error Values	31
5.2	(6,3,2) RS: Detect Only 1 Error	32
5.3	(20,10,5) RS: Big Input with High Error Correction Capability	33
5.4	(20,10,5) RS: Results for Big Input with High Error Correction Capability	34
5.5	Profiling Output	34

List of Tables

2.1	Galois Field $\text{GF}(2^2)$ <i>Table</i>	9
2.2	Galois Field $\text{GF}(2^3)$ <i>Table</i>	10
3.1	Finite Field: <i>alpha_to</i> Table	13
3.2	<i>index_of</i> Table	13

Acronyms

DWTCYCCNT Data Watchpoint and Trace Cycle count

DWT Data Watchpoint and Trace

epi error locator polynomial

psi error evaluator polynomial

ECC Error correction code

GF Galois Field

HAL Hardware abstraction Layer

loc Location

MCU Microcontroller Unit

RS Reed Solomon Code

reg Register

SWD Serial Wire Debug

Bd Baudrate

Abstract

In the project, an extensive Reed Solomon coding is tested out for corrupted communication messages. A flexible model of Reed-Solomon code in a errors and erasure environment is proposed by stating requirements based on the implementation with performance measurement. The model was validated by experimental values. Findings indicate that the proposed model satisfies the primary attributes of a proper reed Solomon coding for a wide range of capacity, imperceptibility, and robustness. The research provides a standing base for further research implementation in a wide range of applications of Reed-Solomon codes.

1 Introduction

In an era of unprecedented digital data transmission and storage, the robust and efficient communication of information has become a paramount concern. Errors inevitably creep into transmitted data due to various factors, including noise, interference, and channel limitations. As a result, the development of error correction codes has emerged as a crucial field in information theory and communication engineering. One such cornerstone of error correction is Reed-Solomon (RS) codes, which have exhibited remarkable performance in combating errors, particularly in scenarios involving binary erasure channels. Reed-Solomon (RS) coding is a widely used error correction technique in information theory and coding theory. It's named after its inventors, Irving S. Reed and Gustave Solomon, and is particularly valuable in scenarios where data transmission or storage is prone to errors, such as in digital communication, data storage devices (like CDs, DVDs, and QR codes), and satellite communication.

1.1 Problem Description

Reed-Solomon (RS) codes are widely employed for error correction in various communication scenarios. These codes are particularly effective in correcting errors and erasures that may occur during data transmission. However, the extent of their effectiveness and their performance characteristics under different channel conditions need to be thoroughly investigated. The primary research problem revolves around understanding the performance limitations and potential enhancements of Reed-Solomon codes for error correction in binary erasure channels.

1.2 Research Objectives

1. Performance Evaluation: This research aims to evaluate the error correction performance of Reed-Solomon codes in the context of binary erasure channels. By conducting systematic experiments, the goal is to quantify the codes' ability to correct errors and erasures across different scenarios.
2. Channel Conditions Impact: Investigate how varying channel conditions, including the number of errors and erasures, impact the effectiveness of Reed-Solomon codes. Determine the threshold beyond which RS codes struggle to provide reliable error correction.
3. Optimization Strategies: Explore potential strategies to enhance the error correction capabilities of Reed-Solomon codes. This includes modifications to the code parameters, optimizing the generator polynomial, and adapting the code to specific use cases.

1.3 Research Design

1.3.1 STM32 Nucleo Board Description:

The STM32 Nucleo board is a popular development platform designed by STMicroelectronics for prototyping and evaluating microcontroller-based projects. The Nucleo boards feature an STM32 microcontroller unit (MCU) at their core, along with various peripherals and connectivity options. These boards are user-friendly and provide an easy way to get started with microcontroller programming and development.

STM32 Nucleo64 M0+ Board:

The STM32 L073RZT6 board is based on the ARM Cortex-M0+ processor, which is a low-power, 32-bit processor architecture. It offers a range of features including GPIO (General Purpose Input/Output) pins, UART (Universal Asynchronous Receiver-Transmitter) ports, SPI (Serial Peripheral Interface) ports, I2C (Inter-Integrated Circuit) ports, and more. The board typically includes an integrated ST-Link programmer and debugger, which allows you to program and debug the MCU using a USB connection. It also provides compatibility with Arduino shields, which allows for additional expansion and connectivity options.

1.3.2 Hardware setup

1. Connect the Nucleo Board: Plug your STM32 Nucleo M0+ board into your computer using a USB cable. This cable provides power to the board and allows for programming and debugging.
2. STM32CubeIDE Installation: STM32CubeIDE is an integrated development environment for STM32 microcontrollers. Install STM32CubeIDE on your computer to create, edit, and debug your C code.
3. Install Required Libraries: In your STM32CubeIDE project, you can use the STM32Cube HAL (Hardware Abstraction Layer) libraries. These libraries provide functions to interact with the various peripherals of the STM32 microcontroller.
4. Toolchain Configuration: STM32CubeIDE comes with a toolchain that includes a compiler, linker, and debugger. Configure your project's toolchain settings according to the STM32 Nucleo M0+ board specifications.
5. Code Implementation: Write your C code in STM32CubeIDE. Utilize the provided HAL libraries to interact with the peripherals of the STM32 microcontroller.
6. Build and Flash: Build your project and generate the firmware binary. Then, use the integrated ST-Link programmer to flash the firmware onto the STM32 Nucleo board.
7. Run and Debug: Use the debugger integrated into STM32CubeIDE to run and debug your code on the Nucleo board. You can set breakpoints, inspect variables, and step through your code to identify and fix issues.

With the hardware setup, libraries, and toolchain configured, you can start implementing your C code on the STM32 Nucleo M0+ board. Remember to refer to the STM32 Nucleo board's datasheet and reference manual for specific pin configurations, memory mappings, and peripheral details.

1.4 Scope

This master project delves into the intricacies of Reed-Solomon codes as applied to the AWGN channel with erasure errors, exploring their theoretical underpinnings, encoding and decoding mechanisms, and practical applications. the project also assess the efficiency of the code implemented.

2 Coding Theory

2.1 Hamming Distance

[?] Hamming distance serves as a fundamental metric to quantify the dissimilarity between two strings of equal length. For instance, consider the strings "11001" and "10010." The Hamming distance between them is 3, as they differ in three positions. In coding theory, Hamming distance plays a vital role in measuring the resilience of error-correcting codes. A code with a higher minimum Hamming distance can detect and correct more errors as in the Reed Solomon coding [2].

2.2 Types of Codes

Various types of error-correcting codes exist, each offering distinct advantages. For instance, Hamming codes are single-error-correcting codes that add redundant bits to a message to detect and correct errors. On the other hand, Reed-Solomon codes[3], the focus of this research, are powerful cyclic codes capable of correcting multiple errors and erasures[4]. These codes are particularly adept at addressing burst errors, which occur when multiple adjacent bits are corrupted.

2.3 Errors

Errors in digital communication result from noise and interference during transmission. For instance, a transmitted "0" might be received as a "1" due to noise. Error-correcting codes aim to rectify such errors by introducing redundancy into the transmitted data. In the case of Reed-Solomon codes, a message is divided into symbols, and additional symbols are added to provide redundancy[2]. This redundancy empowers the receiver to detect and rectify errors.

2.4 Galois Fields

Galois fields, also known as finite fields, form the mathematical underpinning of Reed-Solomon codes[1]. They facilitate arithmetic operations in codes, enabling efficient error correction. In the context of coding theory, Galois fields are essential for computations involving symbols, polynomials, and coefficients. For example, addition and multiplication in Galois fields differ from their conventional counterparts due to the finite nature of the field.

Arithmetic in Finite Field

[1] Performing arithmetic operations in a finite field involves modulo operations on the field's characteristic. For instance, in a binary Galois field, addition is performed modulo 2, resulting in either 0 or 1. Multiplication in finite fields is more intricate, involving polynomial multiplication and reduction using the field's irreducible polynomial.

2.5 Arithmetic of Finite Field $GF(2^2)$

2.5.1 Field Elements

Let's start by defining the field elements of $GF(4)[1]$:

Element	Polynomial Representation
0	0
1	1
α	α
α^2	α^2
$\alpha + 1$	$\alpha + 1$
$\alpha^2 + \alpha$	$\alpha^2 + \alpha$
$\alpha^2 + \alpha + 1$	$\alpha^2 + \alpha + 1$

Here, α is a primitive element of $GF(4)$.

2.5.2 Addition Table

Next, let's create the addition table for $GF(4)[1]$:

+	0	1	α	α^2	$\alpha + 1$	$\alpha^2 + \alpha$	$\alpha^2 + \alpha + 1$
0	0	1	α	α^2	$\alpha + 1$	$\alpha^2 + \alpha$	$\alpha^2 + \alpha + 1$
1	1	0	$\alpha + 1$	$\alpha^2 + \alpha$	α	$\alpha^2 + \alpha + 1$	α^2
α	α	$\alpha + 1$	0	1	$\alpha^2 + \alpha + 1$	α^2	$\alpha^2 + \alpha$
α^2	α^2	$\alpha^2 + \alpha$	1	0	$\alpha^2 + \alpha + 1$	α	$\alpha + 1$
$\alpha + 1$	$\alpha + 1$	α	$\alpha^2 + \alpha + 1$	$\alpha^2 + \alpha$	0	1	α^2
$\alpha^2 + \alpha$	$\alpha^2 + \alpha$	$\alpha^2 + \alpha + 1$	α^2	α	1	0	$\alpha + 1$
$\alpha^2 + \alpha + 1$	$\alpha^2 + \alpha + 1$	α^2	$\alpha + 1$	$\alpha^2 + \alpha$	$\alpha^2 + \alpha + 1$	$\alpha + 1$	0

2.5.3 Multiplication Table

Now, let's create the multiplication table for $GF(4)[1]$:

.	0	1	α	α^2	$\alpha + 1$	$\alpha^2 + \alpha$	$\alpha^2 + \alpha + 1$
0	0	0	0	0	0	0	0
1	0	1	α	α^2	$\alpha + 1$	$\alpha^2 + \alpha$	$\alpha^2 + \alpha + 1$
α	0	α	$\alpha^2 + \alpha$	$\alpha + 1$	$\alpha^2 + \alpha + 1$	1	α^2
α^2	0	α^2	$\alpha + 1$	1	$\alpha^2 + \alpha$	$\alpha^2 + \alpha + 1$	α
$\alpha + 1$	0	$\alpha + 1$	$\alpha^2 + \alpha + 1$	$\alpha^2 + \alpha$	α^2	α	1
$\alpha^2 + \alpha$	0	$\alpha^2 + \alpha$	1	$\alpha^2 + \alpha + 1$	α	α^2	$\alpha + 1$
$\alpha^2 + \alpha + 1$	0	$\alpha^2 + \alpha + 1$	α^2	α	1	$\alpha + 1$	$\alpha^2 + \alpha$

2.5.4 Reduced Forms

The reduced forms of the field elements are as follows[1]:

$$\alpha^2 + \alpha + 1 \equiv 0 \pmod{2}$$

$$\alpha^2 + \alpha \equiv 0 \pmod{2}$$

$$\alpha^2 + 1 \equiv 0 \pmod{2}$$

$$\alpha^2 \equiv 0 \pmod{2}$$

Input Coefficients	Output Coefficients	Polynomial
0	0	0
1	1	1
α	α	α
$\alpha + 1$	α^2	$\alpha^2 + \alpha$

Table 2.1: Galois Field $GF(2^2)$ Table

These are the reduced forms of the elements in $GF(4)$.

2.6 Construction of Finite Field $GF(2^3)[1]$

- **Step 1: Define the Binary Field $GF(2)$**

A finite field of order 2, denoted as $GF(2)$, consists of two elements: 0 and 1. This forms the basis for constructing larger finite fields.

- **Step 2: Find an Irreducible Polynomial**

To construct $GF(8)$, we need an irreducible polynomial of degree 3 over $GF(2)$. An irreducible polynomial is one that cannot be factored into lower-degree polynomials over the same field. Let's choose an example: $p(x) = x^3 + x + 1$.

- **Step 3: Construct the Field's Elements**

In $GF(8)$, each element is represented by a polynomial of degree at most 2 with coefficients from $GF(2)$. The coefficients of the polynomials can be either 0 or 1.

The elements of $GF(8)$ are:

1. 0
2. 1
3. x
4. $x + 1$
5. x^2
6. $x^2 + 1$
7. $x^2 + x$
8. $x^2 + x + 1$

- **Step 4: Define Addition and Multiplication**

Addition and multiplication in $GF(8)$ are performed using polynomial arithmetic, followed by reduction using the irreducible polynomial $p(x)$.

Addition Example:

Let's add $x^2 + 1$ and $x^2 + x$:

$$\begin{aligned}(x^2 + 1) + (x^2 + x) &= (0) \cdot x^2 + (1 + 1) \cdot x + 1 \cdot 1 \\ &= x + 1\end{aligned}$$

Multiplication Example:

Let's multiply x^2 and $x^2 + x + 1$:

$$\begin{aligned}(x^2) \cdot (x^2 + x + 1) &= (0) \cdot x^3 + (1) \cdot x^2 + (1) \cdot x + (1) \cdot 1 \\ &= x^2 + x + 1\end{aligned}$$

• **Step 5: Perform Reduction**

After addition and multiplication, we reduce the resulting polynomials using the irreducible polynomial $p(x)$ to ensure that the results are within $GF(8)$.

Reduction Example:

Let's consider the product $x^2 \cdot (x^2 + x + 1)$ that we obtained earlier: $x^2 + x + 1$. The polynomial is reduced by modulo $p(x)$:

To perform the reduction, perform the polynomial long division:

$$\begin{array}{r|l} x & x^2 + x + 1 \\ & x^2 + x^2 + x \\ & \underline{-(x^2 + x^2)} \\ & x + 1 \\ & \underline{-(x + 1)} \\ & 0 \end{array}$$

The remainder is 0 in $GF(2)$. So, the result after reduction is x in $GF(8)$.

Input Coefficients	Output Coefficients	Polynomial
0	0	0
1	1	1
α	α	α
α^2	α^2	α^2
α^3	α^3	α^3
α^4	α^4	α^4
α^5	α^5	α^5
α^6	α^6	α^6

Table 2.2: Galois Field $GF(2^3)$ Table

By following these steps, one can successfully construct the finite field $GF(8)$ using the irreducible polynomial $x^3 + x + 1$, and we have demonstrated addition, multiplication, and reduction operations within this field using specific examples.

3 Reed-Solomon Coding

3.1 Characteristics of Reed-Solomon codes

At its core, Reed-Solomon coding ($Data[N], Parity[K], error_{detected}[2t]$) is a method of adding redundant information, called parity or check symbols, to the original data before transmission or storage[5]. These additional symbols are calculated in a way that they contain information about the original data, but they're strategically distributed to be able to detect and correct errors that might occur during transmission or storage. [6] The process involves treating the data and the additional parity symbols as coefficients of a polynomial equation.[7] By performing arithmetic operations within a finite field (usually Galois field), the polynomial is generated in such a way that the coefficients of this polynomial represent the original data and the parity symbols.[3] This polynomial is then evaluated at specific points to create the final codeword that is transmitted or stored.

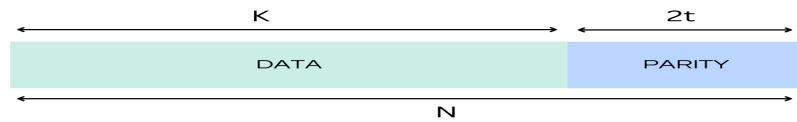


Figure 3.1: Reed Solomon Codeword

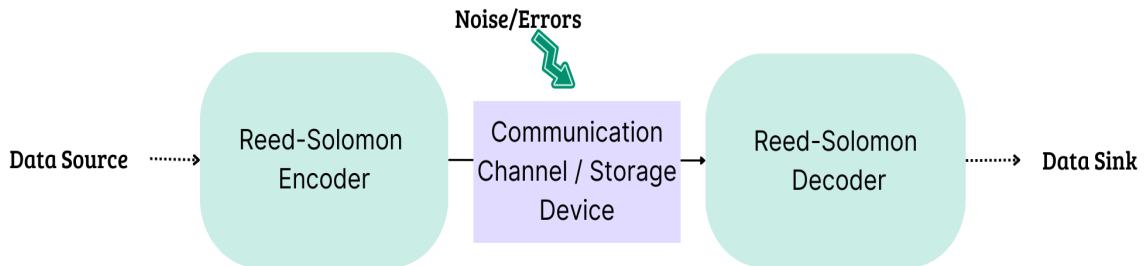


Figure 3.2: (20,10,5) RS: encoder and decoder system

3.1.1 Significance of Reed-Solomon Coding:

1. Robust Error Correction: Reed-Solomon codes are capable of correcting multiple errors within a block of data. This makes them highly effective in scenarios where errors can occur due to noise, interference, or other factors.[8]
2. Efficient[9] and reliable Error Detection: Amongst all ECC methods reed s +olomon code has been found to have.[10]
3. Versatility: Reed-Solomon codes can be tailored to specific error correction needs by adjusting the number of parity symbols and the code's length. This adaptability makes them suitable for a wide range of applications.[3]

4. Low Complexity Decoding: Decoding Reed-Solomon codes involves solving a system of equations, making the process relatively efficient and practical for real-time applications.[4]
5. Strong Error Correction: Reed-Solomon codes are capable of correcting a higher number of errors as Reed-Solomon codes achieve the largest possible code minimum distance for any linear code with the same encoder input and output block lengths[2], making them ideal for scenarios with high error rates.
6. Block-Based Correction: Reed-Solomon codes operate on fixed-size blocks of data, simplifying the decoding process and reducing the latency introduced during error correction.[2]

3.2 Methodology:

3.2.1 Encoding Reed-Solomon Codes

Primitive polynomial

Reed-Solomon codes are defined over a finite field $\text{GF}(2^m)$ generated by a primitive polynomial $pp(x)$ of degree m . This finite field consists of 2^m elements, and the coefficients of all polynomials are binary (0 or 1). The primitive polynomial $pp(x)$ is given by:

$$pp(x) = 1 + \alpha x + \alpha^2 x^2 + \dots + \alpha^{m-1} x^{m-1} + x^m$$

Here, α is a primitive element of the finite field, and x represents the variable of the polynomial.

Generator Polynomial

The generator polynomial $g(x)$ plays a crucial role in encoding data in Reed-Solomon codes. It is formed by multiplying $(x + \alpha^i)$ for i in the range $[1, 2 \cdot t_t]$, where t_t is the number of errors the code can correct. The generator polynomial is expressed as follows:

$$g(x) = (x + \alpha^1)(x + \alpha^2) \dots (x + \alpha^{2t_t})$$

The coefficients of $g(x)$ are calculated based on the coefficients of the primitive polynomial $pp(x)$ and the properties of finite fields.

Finite Field Generation To generate the finite field $\text{GF}(2^m)$, we use the primitive polynomial $pp(x)$ and create lookup tables to represent powers of the primitive element α within the field. These tables are crucial for various calculations in Reed-Solomon coding.

Here are two essential lookup tables:

- **alpha_to[i]**: This table stores values representing α^i in the finite field.
- **index_of[α^i]**: This table stores values representing the index i corresponding to α^i within the finite field.

These tables are generated as follows:

In table 3.1, we start with $i = 0$ and calculate successive powers of α . When $i = m$, we wrap around to 1 to create a cyclic field.

Table 3.1: Finite Field: *alpha_to* Table

i	Value
0	1
1	α
2	α^2
\dots	\dots
$m-1$	$\alpha^{(m-1)}$
m	1 (Wrap-around)

Table 3.2: *index_of* Table

α^i	i
1	0
α	1
α^2	2
\dots	\dots
$\alpha^{(m-1)}$	$m-1$
α^m	0 (Wrap-around)

In table 3.2, we list powers of α and their corresponding indices within the finite field. When α^i reaches α^m , we wrap around to 0 to create a cyclic field.

These lookup tables 3.1 3.2, **alpha_to** and **index_of**, are essential for efficiently performing calculations in Reed-Solomon coding, including encoding and decoding processes. They help in determining the coefficients and properties of the generator polynomial and other critical operations within the finite field $GF(2^m)$.

Encoding Data

The encoder in Reed-Solomon coding plays a crucial role in adding redundancy to the original data. This redundancy allows the code to detect and correct errors that may occur during data transmission or storage, making Reed-Solomon codes an effective solution for error correction in various applications. This process involves generating parity symbols based on the generator polynomial $g(x)$ of the RS code. These parity symbols are added to the original data to form the encoded data, which is then transmitted or stored.

The encoding process involves multiplying the input message polynomial $data(X)$ by the generator polynomial $g(X)$ [3]:

$$c(X) = data(X) \cdot g(X)$$

This operation is performed in the finite field $GF(2^m)$. The output polynomial $c(X)$ contains both the original message symbols and parity symbols.

3.2.2 Decoding Reed-Solomon Codes

The decoder is responsible for the error correction process, attempting to recover the original data from the received codeword, even in the presence of errors.[2]

The key aspects of the decoder output are:

- **Error Detection:** The decoder first checks for the presence of errors in the received codeword. It does so by evaluating the syndromes, which are functions of the received symbols. If any syndromes are non-zero, it indicates the presence of errors.
- **Error Localization:** If errors are detected, the decoder proceeds to locate the positions of these errors within the codeword. This is done by finding the roots of the error locator polynomial, which is derived using the Berlekamp-Massey algorithm.
- **Error Correction:** Once the error positions are determined, the decoder corrects the errors by calculating their values. The error values are then subtracted from the received symbols, allowing the decoder to reconstruct the original data.
- **Output Quality:** The decoder output represents the reconstructed data. The quality of the output depends on the number of errors corrected (t) and the code's design parameters. If the code has a higher error correction capability, it can correct a greater number of errors.

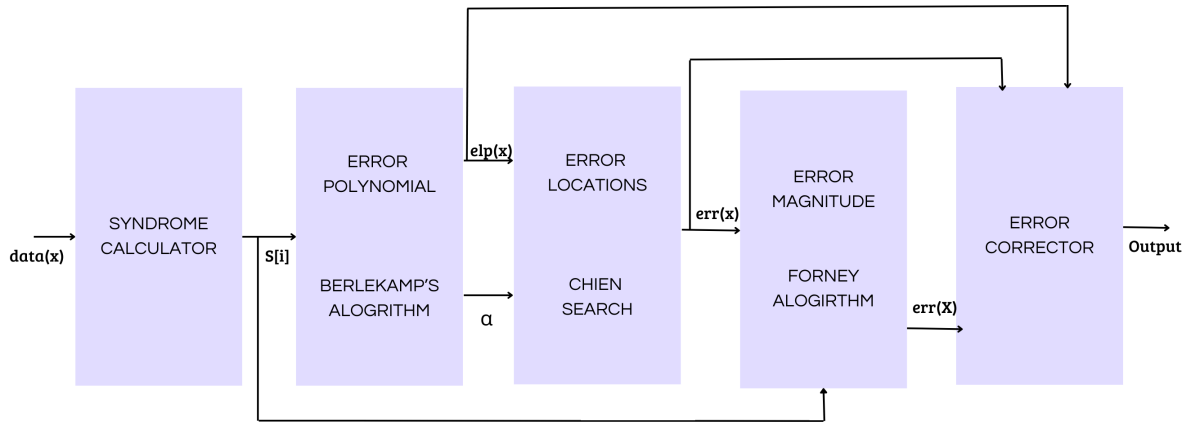


Figure 3.3: Reed Solomon Berlekamp Decoder system

Syndrome Computation

The syndrome calculation in Reed-Solomon coding serves as the initial step for identifying errors in received data. It involves comparing the received symbols with their expected values based on the generator polynomial, producing a syndrome vector that highlights the presence and locations of errors.

- **Syndromes ($s[i]$):** Syndromes are calculated from the received codeword. Syndromes are essentially the evaluations of the received polynomial at certain points. They are computed using the following equation:

$$s[i] = data(\alpha^i), \text{ for } i = 1 \text{ to } (2 \cdot tt)$$

Here, $data(\alpha^i)$ represents the evaluation of the received polynomial at the points α^i in the finite field. The syndromes help identify the error positions.

In summary, this information is crucial for both error detection and correction, enabling the decoder to make informed decisions about how to correct the received data and recover the original information.

Determination of the Error-locator Polynomial and finding roots of the error-locator polynomial

Determining the error-locator polynomial involves employing decoding algorithms like the Berlekamp-Massey algorithm[?] to iteratively refine the polynomial coefficients based on the syndrome vector. Once you have the error locator polynomial, you can find its roots to determine the error locations within the received codeword. This process is a crucial step in the overall Reed-Solomon decoding procedure.

1. **Error Locator Polynomial (elp):** The decoder constructs an error locator polynomial to determine the locations of errors in the received codeword. The Berlekamp-Massey algorithm is typically used to find $elp(X)$.

$$elp(X) = 1 + \sum_{i=1}^{l[u]} (elp[i] \cdot X^{(loc[i])})$$

Here,

- $elp[i]$ represents the coefficients of the error locator polynomial.
- $loc[i]$ represents the error locations (positions where errors occurred).
- $l[u]$ is the degree of the error locator polynomial.

Calculation of Error Values

Calculating the error values in a Reed-Solomon code involves determining the magnitude of the errors at specific error locations within the received codeword. This step is often done using a formula known as Forney's formula, which uses the error locator polynomial and the error evaluator polynomial.

1. **Error Magnitude Calculation($err(X)$)**

Error evaluator polynomial helps calculate the error value at that specific location. The error evaluator polynomial is derived from the syndrome polynomial and the error locator polynomial.

The error evaluator polynomial $err(x)$ is computed to determine the error values at error positions. It involves solving for the values of $err(x)$ using the roots and syndromes.

Once error positions are found, the decoder constructs an error evaluator polynomial to calculate the values of errors at those positions:

$$err(X) = \sum_{i=0}^{l[u]} (err[i] \cdot X^{(loc[i])})$$

Where: - $err[i]$ represents the coefficients of the error evaluator polynomial.

The Chien search algorithm is used to find the roots of the error locator polynomial, and when a root is found, this formula is used to calculate the corresponding error value at that root position. If the error locator polynomial has roots at certain positions, it means there are errors at those positions, and this formula helps determine the values of those errors.

The formula is given by:

$$error_value = syndrome_value / error_evaluator_value$$

Here,

- *syndrome_vvalue*: The value of the syndrome polynomial at the error location.
- *error_eevaluator_vvalue*: The value of the error evaluator polynomial at the error location.

Error Correction

With error positions and their magnitudes known, the decoder corrects the received codeword by subtracting the error polynomial from it.

Output Correction or Flagging

Depending on the number of errors and whether they are correctable, the decoder can either:

- Correct the errors and output the corrected message.
- Output an error flag if there are too many errors to correct.
- Output the received data as is if there are no errors.

These mathematical and algorithmic steps together enable Reed-Solomon codes to detect and correct errors in the received data, making them robust for various applications, especially in data storage and transmission.

4 Implementation

The goal is to demonstrate the meticulous implementation of the Reed-Solomon Error Correction Code (ECC) and its profiling on the STM32L073RZT6[11] microcontroller, harnessed through STM32CubeIDE[12]. Reed-Solomon ECC, renowned for its reliability in data transmission and storage, will be our tool for ensuring data integrity in this microcontroller environment. Additionally, we'll employ profiling techniques to gain insights into the code's performance, resource utilization, and execution timing.

4.1 ARM CORTEX M0+ Microcontroller

The STM32L073RZT6 microcontroller, leveraging the ARM Cortex-M0+ architecture, is primarily tailored for energy-conscious embedded systems[13]. It belongs to the STM32L0 family renowned for its remarkable power efficiency and minimal power consumption traits[11]. While it stands out in terms of energy frugality, it does come with certain hardware and feature trade-offs that can impact code profiling in comparison to its STM32 counterparts.

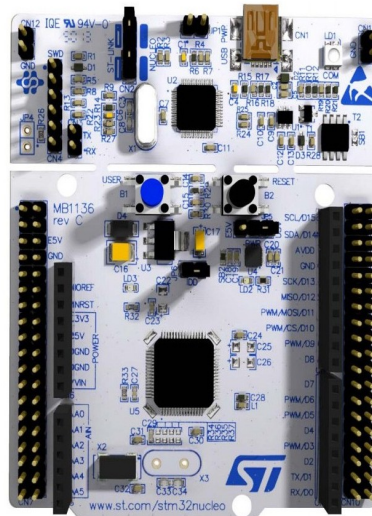


Figure 4.1: STM32 L073RZT6

The microcontroller in the STM32L0 series provides an array of low-power modes, including Stop, Standby, and Sleep modes, enabling the microcontroller to curtail power utilization during periods of inactivity or low demand.[11] This capability is particularly crucial for battery-dependent or power-sensitive applications. The Cortex-M0+ core is distinguished for its meager active power consumption, rendering it apt for tasks that necessitate processing prowess while maintaining minimal power usage. This microcontroller also incorporates energy-efficient peripherals such as low-power timers, an ultra-low-power RTC (Real-Time Clock), and low-power UART/I2C/SPI interfaces. These features further bolster energy efficiency.

4.1.1 Profiling Limitations:

Despite the power and energy efficient feature of the ARM CORTEX M0+ micro-controller.

There are several limitations when running a quality profiling code.

A subset of ARM Cortex-M0+ cores offers restricted performance counters, which play a pivotal role in advanced profiling and optimization endeavors. These counters gauge various aspects of code execution, and their scarcity can curtail profiling capabilities.

In comparison to other ARM Cortex-M cores like Cortex-M3 and Cortex-M4, the Cortex-M0+ core exhibits lower power consumption but also lower performance. It boasts a limited instruction set and lacks hardware components such as hardware floating-point units or SIMD (Single Instruction, Multiple Data) instructions[14]. This limitation can hinder the precision and versatility of code profiling and execution time measurement, particularly for complex algorithms.

The STM32L073RZT6 microcontroller, based on the ARM Cortex-M0+ core, does not include the Data Watchpoint and Trace (DWT) cycle counter (DWT_CYCCNT) as a hardware feature[14]. Since the STM32L073RZT6 uses the Cortex-M0+ core, one does not have direct access to the (DWT_CYCCNT) register for cycle-accurate profiling and timing measurements. The STM32L0 microcontrollers lack advanced debugging features like trace capabilities, which are invaluable for in-depth code profiling and real-time debugging.

However, in this project, we can use other methods to measure execution time and profile your code, such as using timers[11] and manual instrumentation as previously discussed.

In summation, the STM32L073RZT6 microcontroller, founded on the ARM Cortex-M0+ core, constitutes a stellar choice for energy-efficient embedded systems with stringent power constraints. Nonetheless, for applications necessitating extensive code profiling and real-time performance optimization, it may not emerge as the optimal selection due to its reduced performance and limited hardware profiling capabilities. But this project requires energy-efficient and cost-effective specific applications hence we chose the STM32 ARM Cortex M0+ microcontroller.

4.2 STM32CUBEIDE:

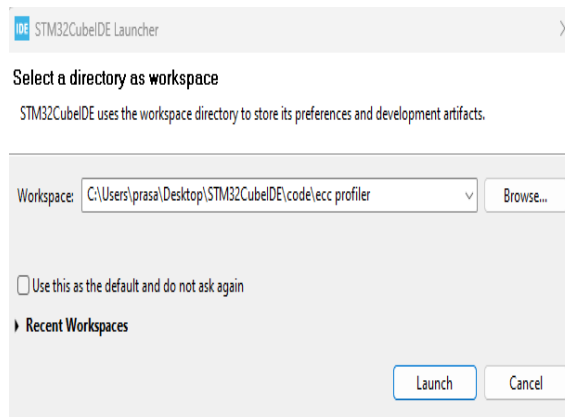
4.2.1 Configure STM32CubeIDE:

Open STM32CubeIDE and make a new workspace for all your STM32 M0+ files. Create a new STM32 project. If you're starting a new project, STM32CubeIDE will guide you through the process of configuring the microcontroller and project settings.

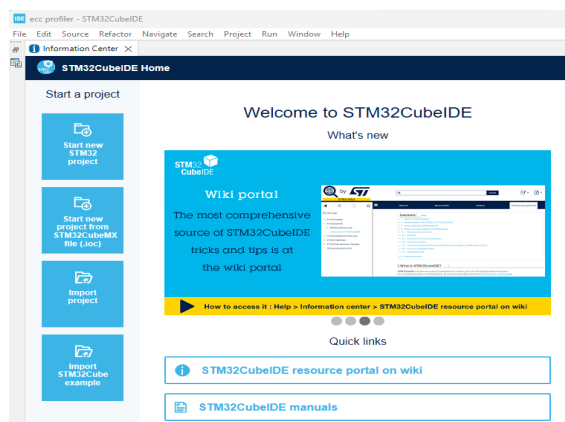
Choose The appropriate board, STM32 L073RZT6 microcontroller and select a project name and click finish. (here: ecc profiler)

4.2.2 System Debug and Serial Wire Debug (SWD) Configuration

The STM32L073RZT6 microcontroller's debugging capabilities are pivotal in ensuring that ECC implementation operates flawlessly. Hence we use the Serial Wire Debug (SWD) interface provided by STM32CubeIDE to inspect the microcontroller's inner workings(Fig 4.4). SWD allows us to step through the code, set breakpoints, and monitor the variables' states, ensuring that the ECC logic operates as expected[11]. This debugging infrastructure will be our watchful eye, ensuring the reliability and accuracy of our ECC implementation.

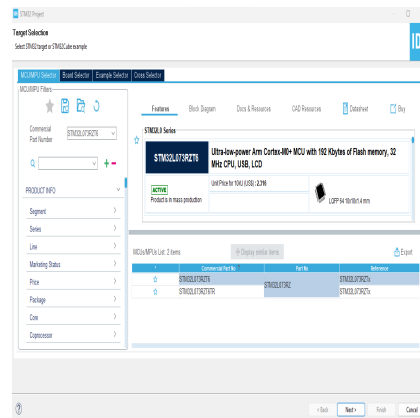


(a) Create workspace

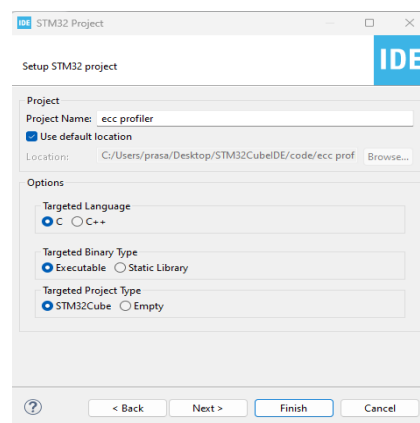


(b) Create new STM32 project

Figure 4.2: Start new project on STM32CubeIDE 1.12.1



(a) Select STM32 L073RZT6 board from serial number



(b) Create a C executable with a suitable project name

Figure 4.3: Select Board and Create C project

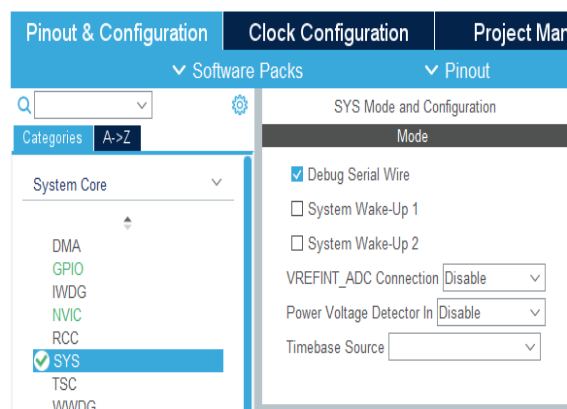
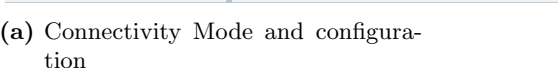


Figure 4.4: System Configuration with debug serial wire

The parameters are set to :



operations within the ECC implementation[11]. We carefully select the operating mode, set the prescaler, and define the auto-reload value to align TIM2's capabilities with the requirements of our ECC algorithm. This meticulous timing control ensures that our ECC calculations occur with the desired precision.

1. Prescaler (TIMx-¿PSC): The prescaler divides the timer's input clock frequency. The purpose of the prescaler is to scale down the input clock to an appropriate value for your application.[15]

- a) Resolution: The prescaler affects the timer's resolution. A lower prescaler value gives you finer resolution, but it decreases the timer's range. A higher prescaler value increases the timer's range but reduces resolution.

Example: If your input clock is 16 MHz and you set the prescaler to 15, you effectively divide the clock by 16 ($16 \text{ MHz} / 16 = 1 \text{ MHz}$). This means the timer will increment every microsecond, giving you a 1 μs resolution.

2. Period (TIMx-¿ARR): The period is the value at which the timer counter resets and generates an update event. It determines the maximum count value the timer can reach before it rolls over and restarts from zero.[15]

- a) Range: The period value sets the upper limit for the timer. If the timer reaches this value, it triggers an update event and resets. The period directly affects the maximum time interval you can measure before the timer overflows.

Example: If your timer is configured for a 16-bit counter (common for many STM32 timers) and you set the period to 65535, the timer will count from 0 to 65535 before rolling over, giving you a maximum interval of 65535 timer clock cycles.

4.2.5 Clock Configuration (HSI16)

Clock accuracy is paramount when dealing with error correction and profiling. Our choice of the High-Speed Internal (HSI16) clock source (Fig 4.6) for the STM32L073RZT6 microcontroller is underpinned by its stability and precision[16]. A precise clock is indispensable for profiling, as it enables us to measure and analyze execution times with utmost accuracy. Configuring HSI16 effectively ensures that our profiling data is reliable and aids in optimizing the ECC implementation's performance.

The STM32L0 Series features a simple clock factor-4 divider, associated with the HSI clock source, making the HSI the effective source of either the 16 or the 4 MHz. As a result, the STM32L0 is much more efficient in certain applications requiring UART speeds higher than 9600 Bd.[17]

4.2.6 Project

The c code is then generated with all hardware and software configuration for STM32 L0 board, then proceed to work on the RS code in the project files(Fig 4.7). The mathematical foundation of RS code is translated into a C code tailored for the STM32 L0 series microcontroller. The implementation includes encoding and decoding functions, optimized data structures, and error-checking mechanisms. This ECC code becomes the guardian of data integrity, allowing us to recover from errors and ensuring the STM32 L0 series microcontroller's reliability in data communication.

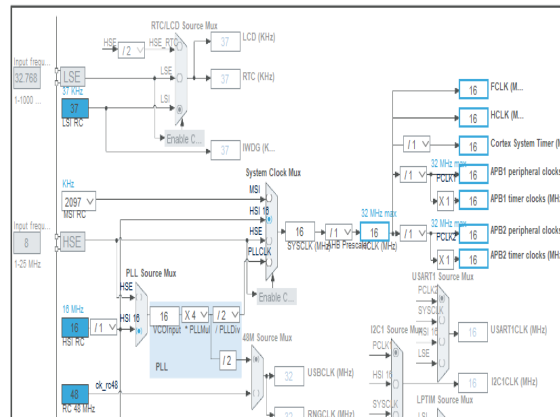


Figure 4.6: Clock setting

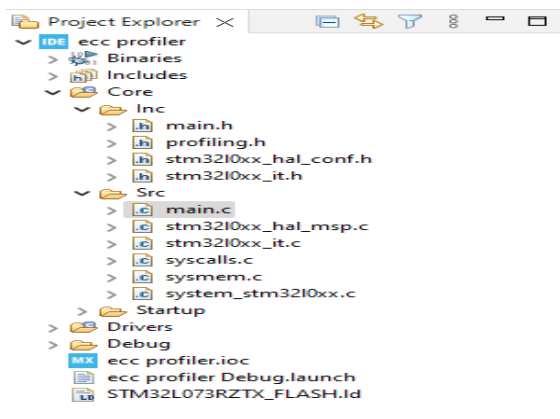


Figure 4.7: Configure main.c

The Reed Solomon code (GitHub[18]) is well-documented with comments explaining the purpose of each function and variable.

The C code provided implements a Reed-Solomon error correction algorithm. This document explains each of the functions used in the implementation and provides details on their functionality and parameters.

4.3 Functions

4.3.1 Check Multiple Inputs

The program performs Reed-Solomon error correction for three different scenarios with different parameters. Each scenario represents a specific Reed-Solomon code and prints the original data, encoded data, and decoded data.

Here's a brief overview of what the code INPUT(int count) does:

This function based on count value sets the following parameters for the Reed-Solomon code. It helps define the characteristics of the code, including Galois field size and error correction capability.

Encoder and Decoder for (10, 4, 6) Reed-Solomon Code

- Parameters: $m = 4$, $N = 10$, $parity = 6$
- Encoding: Simulates at least 3 error values in the encoded data.
- Decoding: Detects and corrects the errors.

Encoder and Decoder for (6, 3, 2) Reed-Solomon Code

- Parameters: $m = 4$, $N = 6$, $parity = 2$
- Encoding: Simulates 1 error value in the encoded data.
- Decoding: Detects and corrects the error.

Encoder and Decoder for (20, 10, 5) Reed-Solomon Code

- Parameters: $m = 4$, $N = 20$, $parity = 5$
- Encoding: Simulates a high number of errors in the encoded data.
- Decoding: Corrects errors due to the high error correction capability.

4.3.2 DECLARATION(int mm, int nn)

This function declares the size of arrays alpha_to, index_of, and gg based on the provided values of mm and nn in order to efficiently manage memory space. Note that this function is not used in the main code.

4.3.3 ENCODE_RS()

This function encodes the input message in a systematic way using the generator polynomial and produces parity symbols.[18]

The *rs_encoder()* function encodes the input data using the generator polynomial and stores the resulting code word.

1. Initializing the Buffer:

- **data[]**: An array containing the symbols to be encoded. These symbols are treated as coefficients of a polynomial.
- **bb[]**: An array that will store the encoded symbols (parity symbols) also represented as polynomial coefficients.

The code initializes the **bb[]** array with zeros for the first **nn - kk** elements. These elements will eventually store the parity symbols.

2. Encoding Loop: This is the main loop responsible for encoding the input data. It iterates through each symbol in the data array in reverse order ($i = k - 1; i \geq 0; i -$). For each symbol, it calculates the corresponding feedback value based on the current symbol and the previous parity symbol.

3. Calculating Feedback: The variable feedback is computed by XORing the current **data[i]** with the most significant element of **bb[]** (i.e., **bb[nn-kk-1]**), and then using the **index_of** array to find the index of this XOR result in the **alpha_to** array. This index corresponds to the exponent of the polynomial form (corresponding index in the Galois field).

Based on the feedback value, the code performs polynomial multiplication and addition operations to update the **bb[]** array. This step involves the use of the **gg[]** array to specify the connections between elements of the feedback shift register.

If the feedback value is -1 (indicating an issue in the calculation), it handles this by shifting the values in the **bb[]** array to the right and assigning **bb[0]** as 0.

4. Parity Symbol Calculation: If the feedback value is not -1 (indicating that the feedback symbol is non-zero), the loop generates new parity symbols based on the generator polynomial **g[]**. It loops through the elements of **g[]** and updates each element in the buffer **b[]** based on the feedback value and the generator polynomial.

If the feedback value is -1 (indicating an issue in the calculation), it handles this by shifting the values in the **bb[]** array to the right and assigning **bb[0]** as 0.

5. Shifting Parity Symbols: After calculating new parity symbols, the function shifts the existing parity symbols in the buffer to the right. This is done to accommodate the new parity symbol at the beginning of the buffer.

6. Loop termination: After the loop completes, the **bb[]** array contains the encoded message in polynomial form. This array represents the codeword with both message symbols and parity symbols.

This code essentially implements polynomial multiplication and addition operations in a systematic way to encode the input data into a Reed-Solomon codeword.

The **alpha_to**, **index_of**, and **gg** arrays play key roles in this process by defining the algebraic properties of the finite field used in Reed-Solomon coding.

4.3.4 DECODE_RS()

This function decodes received symbols to correct errors if they exist. It uses the Berlekamp iteration algorithm to find the error location polynomial and corrects the received symbols.[18]

1. Introducing Errors and Erasures: The code then simulates errors and erasures in the received codeword by modifying specific positions as provided by the user.
2. Decoding: The `rsdecoder()` function attempts to decode the received code word to recover the original data. It uses the Berlekamp-Massey algorithm to correct errors and erasures.
1. Initialization: Initialize arrays and variables required for the algorithm, such as arrays to hold discrepancy, error locator polynomial (elp), psi (error evaluator polynomial), and reg (shift register for error locator polynomial calculation).

- `int elp[nn-kk+2][nn-kk]`: An array to store the error location polynomial.
- `int d[nn-kk+2]`: An array to store discrepancies.
- `int l[nn-kk+2]`: An array to store the degrees of the error location polynomial.
- `int u_lu[nn-kk+2]`: An array to store the difference between the step number and the degree of the error location polynomial.
- `int s[nn-kk+1]`: An array to store syndromes.
- `int count = 0`: A counter to keep track of the number of errors corrected.
- `int syn_error = 0`: A flag to indicate if syndromes are non-zero, implying errors.
- `int root[tt], loc[tt], z[tt+1], err[nn], reg[tt+1]`: Arrays to store intermediate values during error correction. Set the first element of the shift register (`reg[0]`) to 1 as a starting point.

2. Main Berlekamp-Massey Loop: The algorithm uses the Berlekamp iterative algorithm to compute the error location polynomial (elp). This polynomial helps identify the positions of errors in the received data. It iterates over the received symbols (received parity and data).
3. Error Locator Polynomial: For each received symbol, calculate the discrepancy between the received symbol and the computed value based on the error locator polynomial. If the discrepancy is nonzero, and the degree of the error locator polynomial (elp) (*deg_{lambda}*) needs to be increased, and the error evaluator polynomial (psi).

After finding the error locator polynomial, we use Chien's search algorithm to find the locations of errors directly from the roots of the error locator polynomial at various points in the fields. This can be more efficient than calculating the roots individually. These roots represent the error locations in the received codeword.

The algorithm terminates when the degree of the error location polynomial exceeds `tt` (the number of correctable errors) or when it has iterated through all possible discrepancies and proceeds with error correction.

4. Error Correction: It finds the roots of the error location polynomial to determine the error locations and their inverse roots. It then forms a polynomial $z(x)$ and evaluates errors at the locations given by error location numbers. If the number of errors located matches the degree of the error location polynomial, the algorithm corrects the errors.

Once the error locations are determined, the code calculates error values using the values of the error location polynomial and the syndromes.

The algorithm computes the error values and applies them to correct the errors.

5. Output:

- a) If errors are corrected, the algorithm returns the corrected data.
 - b) If errors cannot be corrected (degree of `elp` \geq `tt`), it outputs the received data without correction.
 - c) If there are no syndromes (no errors detected), it outputs the received data as it is.
6. Conditions: Throughout the algorithm, there is a conversion between index form (powers of alpha) and polynomial form (coefficients). This conversion is necessary for various calculations.

The algorithm terminates when the degree of the error location polynomial exceeds `tt` or when it has iterated through all possible discrepancies.

The algorithm can optionally return error flags to the calling routine if desired.

This decoding algorithm efficiently corrects errors in Reed-Solomon codes, taking into account the number of correctable errors (`tt`) and providing flexibility for handling various error scenarios. It is designed to work with both systematic and non-systematic encoding, making it a versatile tool for error correction in data communication and storage systems.

4.3.5 Profiling

Profiling is a crucial aspect of software development, especially in embedded systems where efficiency and timing constraints are essential. The provided code includes profiling functions to measure the execution time of various parts of the program.[19] The profiling output provides insights into the code's performance.

PROFILING_START(const char *profile_name)

This function starts profiling the execution time of the code and records the current time. It starts the profiling and records the current time using `HAL_GetTick()` from the microcontroller's HAL library. The `profile_name` parameter specifies the name of the profiling sequence.

```
void PROFILING_START(const char *profile_name)
{
    prof_name = profile_name;
    event_count = 0;

    // Initialize HAL or CMSIS for your microcontroller
    //HAL_Init();

    //Configure SysTick for 1 us resolution
    //HAL_SYSTICK_Config(SystemCoreClock / 1000000);
    //HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);
```

```

    time_start = HAL_GetTick();
}

```

PROFILING_EVENT(const char *event)

This function records profiling events by capturing the time elapsed since the start of profiling. It records the event name and the time taken for each event.

```

void PROFILING_EVENT(const char *event)
{
    if (event_count == _PROF_STOPPED)
        return;

    if (event_count < MAXEVENTCOUNT)
    {
        time_event[event_count] = HAL_GetTick() - time_start;
        event_name[event_count] = event;
        event_count++;
    }
}

```

There are 3 main profiling events recorded and timed. These events include encoding, decoding, and overall code execution time for the entire Reed-Solomon coding process. In the code, events are named to describe their purposes, such as error detection, correction, encoding, and decoding.

PROFILING_STOP()

This function stops profiling and prints the results of the profiling events, including the time elapsed between events and event names.

```

void PROFILING_STOP(void)
{
    int32_t timestamp;
    int32_t delta_t;
    //int32_t tick_per_1us = SystemCoreClock / 1000000;
    int32_t time_prev = 0;
    const char* targetString = "Runtime ECC Code";

    if (event_count == _PROF_STOPPED)
    {
        DEBUG_PRINTF("\r\nWarning: PROFILING_STOP WITHOUT START.\r\n");
        return;
    }

    DEBUG_PRINTF(" Profiling \"%s\" sequence: \r\n\n"—Event—————
                ————|—timestamp—|———delta_t———\r\n\n",
                prof_name);
}

```



```
for (int i = 0; i < event_count; i++)
{
    timestamp = time_event[i];
    //timestamp = (time_event[i] - time_start) / tick_per_1us;

    if (strncmp(event_name[i], targetString, strlen(targetString))
        == 0)
    {
        delta_t = (i != 2) ? timestamp - time_event[i - 3] :
                        timestamp;
    }
    else
    {
        delta_t = timestamp - time_prev;
    }

    time_prev = timestamp;

    DEBUG_PRINTF("%-70s:%9lu  s  | +%9lu  s \r\n", event_name[i],
        (unsigned long)timestamp, (unsigned long)delta_t);
}

DEBUG_PRINTF("\r\n");
event_count = _PROF_STOPPED;
}
```

Profiling results include the timing information for each event, measured in microseconds (μs). The timing data helps assess the efficiency of various code segments and identify potential bottlenecks.

Overall, the profiling output offers valuable insights into the timing behavior of the Reed-Solomon coding process on the specific hardware platform STM32 L073RZT6 or ARM Cortex M0+ microcontroller. It helps developers optimize the code for performance and resource utilization, ensuring reliable error correction within real-time constraints.

4.4 Print Results:

The received message, corrected message, and other relevant information are printed on the STM32CUBEIDE serial console.

Figure 4.7, Figure 4.8, and Figure 4.9 show how to open a serial console terminal from STM32CubeIDE.



Figure 4.8: Open serial console of STM32CubeIDE

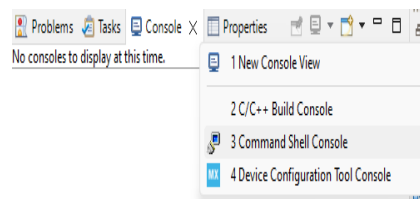


Figure 4.9: Open serial console of STM32CubeIDE

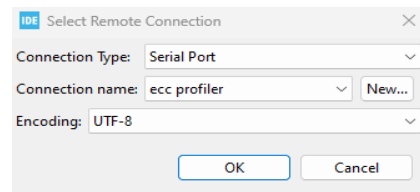


Figure 4.10: Console connection

Please note: The Reed-Solomon code implementation in this project is quite extensive, and the code itself includes several comments that provide explanations for its different parts.

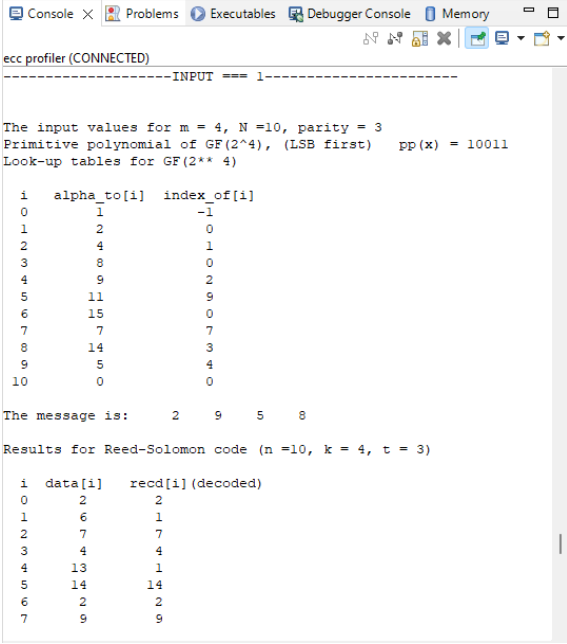
5 Result and Analysis

The results of my implementation, including sample inputs and outputs and the effectiveness of error correction by comparing the original data with the decoded data are analyzed. • The program also outputs the performance of your Reed-Solomon coding in terms of error correction capability. The data (introduced with fixed errors/erasures), and the decoded data are able to be printed to check capability.

The output of the program provides information about the primitive polynomial and constructed Galois Field and corresponding generated polynomial.

Here's an example of what the output might look like:

For input 1:



```
ecc profiler (CONNECTED)
-----INPUT === 1-----

The input values for m = 4, N =10, parity = 3
Primitive polynomial of GF(2^4), (LSB first)  pp(x) = 10011
Look-up tables for GF(2** 4)

i  alpha_to[i]  index_of[i]
0      1         -1
1      2          0
2      4          1
3      8          0
4      9          2
5     11          9
6     15          0
7      7          7
8     14          3
9      5          4
10     0          0

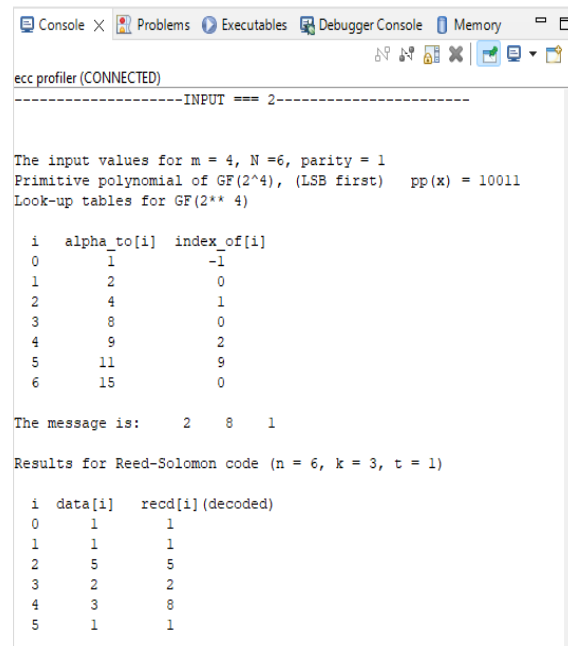
The message is:      2      9      5      8

Results for Reed-Solomon code (n =10, k = 4, t = 3)

i  data[i]  recd[i] (decoded)
0      2         2
1      6         1
2      7         7
3      4         4
4     13         1
5     14        14
6      2         2
7      9         9
```

Figure 5.1: (10,4,6) RS: Detect and Correct at Least 3 Error Values

For input 2:



```

ecc profiler (CONNECTED)
-----INPUT === 2-----

The input values for m = 4, N =6, parity = 1
Primitive polynomial of GF(2^4), (LSB first)  pp(x) = 10011
Look-up tables for GF(2** 4)

i   alpha_to[i]  index_of[i]
0       1         -1
1       2          0
2       4          1
3       8          0
4       9          2
5      11          9
6      15          0

The message is:      2      8      1

Results for Reed-Solomon code (n = 6, k = 3, t = 1)

i   data[i]   recd[i] (decoded)
0       1         1
1       1         1
2       5         5
3       2         2
4       3         8
5       1         1
  
```

Figure 5.2: (6,3,2) RS: Detect Only 1 Error

For input 3:

```

ecc profiler (CONNECTED)
-----INPUT === 3-----

The input values for m = 4, N =20, parity = 5
Primitive polynomial of GF(2^4), (LSB first)  pp(x) = 10011
Look-up tables for GF(2** 4)

i   alpha_to[i]  index_of[i]
0       1         -1
1       2         15
2       4         16
3       8         12
4       9         17
5      11          9
6      15         13
7       7          7
8      14         18
9       5         19
10      10         10
11      13          5
12       3         14
13       6         11
14      12          8
15       1          6
16       2          0
17       4          0
18       8          0
19       9          0
20      -1          0

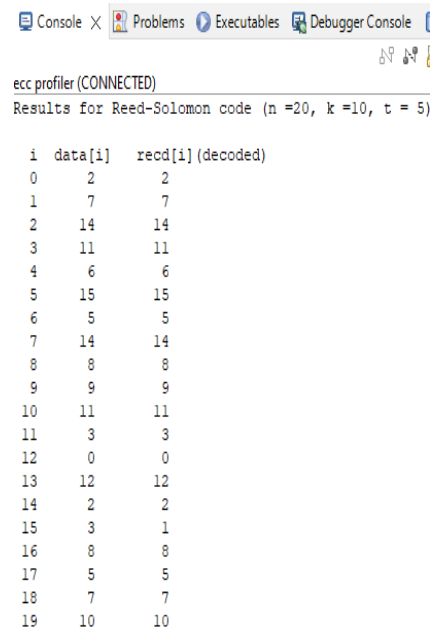
The message is:  11   3   0  12   2   1   8   5   7  10

```

Figure 5.3: (20,10,5) RS: Big Input with High Error Correction Capability

Profiling result:

Note: Additionally all the steps involved in the encoding and decoding process can be out-putted by un-commenting the parts in the code.



```

ecc profiler (CONNECTED)
Results for Reed-Solomon code (n =20, k =10, t = 5)

i  data[i]  recd[i] (decoded)
0   2       2
1   7       7
2  14      14
3  11      11
4   6       6
5  15      15
6   5       5
7  14      14
8   8       8
9   9       9
10  11      11
11  3       3
12  0       0
13  12      12
14  2       2
15  3       1
16  8       8
17  5       5
18  7       7
19  10      10

```

Figure 5.4: (20,10,5) RS: Results for Big Input with High Error Correction Capability

```

Profiling "RS ECC startup timing" sequence:

--Event-----|---timestamp---|---delta_t---

(10,4,6) Encoder: Detect and Correct at Least 3 Error Values      :    55 µs | +    55 µs
(10,4,6) Decoder: Detect and Correct at Least 3 Error Values      :    59 µs | +     4 µs
Runtime ECC Code                                                  :    88 µs | +    88 µs
(6,3,2) Encoder: Detect Only 1 Error                             :   131 µs | +    43 µs
(6,3,2) Decoder: Detect Only 1 Error Value                       :   134 µs | +     3 µs
Runtime ECC Code                                                  :   155 µs | +    67 µs
(20,10,5) Encoder: Big Input with High Error Correction Capability :   240 µs | +    85 µs
(20,10,5) Decoder: Big Input with High Error Correction Capability :   246 µs | +     6 µs
Runtime ECC Code                                                  :   296 µs | +   141 µs

```

Figure 5.5: Profiling Output

6 Conclusion

This research project embarked on a journey to explore and implement Reed-Solomon (RS) coding on the STM32 ARM Cortex-M0+ microcontroller platform. The research objectives revolved around evaluating the performance of Reed-Solomon codes, understanding the impact of varying channel conditions, and exploring potential optimization strategies to enhance error correction capabilities.

The Reed Solomon Code implemented systematically evaluates the error correction performance of Reed-Solomon codes, specifically in the challenging environment of errors and erasures. Through rigorous experimentation, we sought to quantify the ability of RS codes to correct errors and erasures across a spectrum of scenarios.

The implementation further delved into the intricate relationship between channel conditions and the effectiveness of Reed-Solomon codes. It investigated and studied how variations in the number of errors and erasures impact the reliability of RS codes. By identifying thresholds beyond which RS codes struggle to provide reliable error correction, we sought to gain deeper insights into their limitations.

The project also focuses on optimizing the error correction capabilities of Reed-Solomon codes. This included examining the potential execution time of the encoder and decoder for each code parameter, and fine-tuning the allocated size of the generator polynomial. By doing so, we aimed to uncover opportunities to enhance the performance of RS coding in practical applications.

The research was conducted on the STM32CubeIDE platform for STM32 microcontroller, a versatile development board provided by STMicroelectronics. Specifically, we used the STM32 L0 series board, which is based on the ARM Cortex-M0+ processor architecture. This choice was driven by the energy-efficient design of the Cortex-M0+ processor. Despite its many advantages, it's essential to acknowledge that the STM32 L0 series microcontroller also comes with certain profiling and constraints due to its energy-efficient build. The performance evaluation was developed to work with this platform by considering all these limitations when designing and implementing this project.

In conclusion, this research study on implementing Reed-Solomon coding on the STM32 ARM Cortex-M0+ microcontroller platform has contributed valuable insights into the practical use of RS codes in error correction scenarios. The outcomes of this research will aid in improving the reliability and performance of embedded systems that rely on Reed-Solomon codes for error correction.

A Appendix

Bibliography

- [1] Wikipedia contributors. Finite field, 2023. [Online; accessed 5-September-2023].
- [2] Bernard Sklar. *Digital Communications: Fundamentals and Applications*. Prentice-Hall PTR, 2001.
- [3] Wikipedia contributors. Reed-solomon error correction, 2023. [Online; accessed 4-September-2023].
- [4] Lilian Atieno, Jonathan Allen, Dennis Goeckel, and Russell Tessier. An adaptive reed-solomon errors-and-erasures decoder. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, page 150–158, New York, NY, USA, 2006. Association for Computing Machinery.
- [5] W. Cary Huffman and Vera Pless. *BCH and Reed-Solomon Codes*, page 168–208. Cambridge University Press, 2003.
- [6] Stephen B Wicker and Vijay K Bhargava. *Reed-Solomon Codes and Their Applications*. John Wiley & Sons, 1999.
- [7] Joel Sylvester. Reed solomon codes. *Elektrobit*, January, 2001.
- [8] Priyanka Shrivastava and Uday Pratap Singh. Error detection and correction using reed solomon codes. *International Journal of Advanced Research in Computer Science and Software Engineering*, 3(8), 2013.
- [9] Sheryl Howard, Christian Schlegel, and K. Iniewski. Error control coding in low-power wireless sensor networks: When is ecc energy-efficient? *EURASIP J. Wireless Comm. and Networking*, 2006, 12 2006.
- [10] Imad Ez-zazi, Mounir Arioua, Ahmed el Oualkadi, and Younes el Assari. Performance analysis of efficient coding schemes for wireless sensor networks. In *2015 Third International Workshop on RFID And Adaptive Wireless Sensor Networks (RAWSN)*, pages 42–47, 2015.
- [11] Stm32l073rz datasheet.
- [12] Stm32cubeide.
- [13] Arm cortex-m0+.
- [14] Wikipedia contributors. Arm cortex-m, 2023. [Online; accessed 5-September-2023].
- [15] Shawn Hymel. Getting started with stm32 - timers and timer interrupts.
- [16] How to calibrate an stm32l0xx internal rc oscillator.
- [17] Minimization of power consumption using lpuart for stm32 microcontrollers.

- [18] Aditi Prasad. Reed solomon code c code.
- [19] Sergey Bashlayev and Stan Verschuuren. Stm32 profiler.