

User's Manual for LinCodeWeightInv Library

Maria Pashinska-Gadzheva, Iliya Bouyukliev
Institute of Mathematics and Informatics, Bulgarian Academy of Sciences

Abstract: The Linear Codes Weight Invariant Library (LinCodeWeightInv) is a C/C++ library for fast computation of the weight distribution and connected parameters of linear codes over finite fields using Extended Vector Registers (SSE4.1, AVX2, AVX512, NEON for ARM architectures). The library includes six basic end user functions that include calculations of the weight distribution, the minimum distance d and the number of codewords with a given weight among others. The presented Sample program can be used to execute calculation with all functionalities with two different input methods (reading data from file and generating random test data). It also has a functionality for correctness and runtime testing with the developed vectorization options and without vectorization. The current manual gives information about the compilation and installation process, description of the included Sample program and a file documentation of the presented `.h` files that contain the main function declarations.

Acknowledgements: The work of the first author was partially supported by the the Centre of Excellence in Informatics and ICT established under the Grant No BG05M2OP001-1.001-0003, financed by the Science and Education for Smart Growth Operational Program and co-financed by the European Union through the European Structural and Investment funds. The work of the second author has been accomplished with the financial support of the Bulgarian MES by the Grant No. DO1-98/26.06.2025 for NCHDC – part of the Bulgarian National Roadmap on RIs.

1 Introduction	5
1.1 Introduction	5
1.1.1 Field Elements Representation	5
2 Compilation and Installation	7
2.1 Compilation	8
2.1.1 Setting up IDE project with CMake	8
2.1.2 Compiling with IDE	10
2.1.3 Compiling on Linux OS	10
2.1.4 Compiling for AVX512	12
2.1.5 Compilation using the included Makefiles	12
2.1.6 Compilation using specific instruction set on x86 architecture	12
2.2 Installation	13
2.2.1 Using library in personal projects	13
2.3 Tested platforms	13
3 Sample and testing executable programs	15
3.1 Sample program	15
3.2 Testing executable programs	20
4 File Documentation	23
4.1 testDriver.h File Reference	23
4.1.1 Function Documentation	24
4.1.1.1 test_drive()	24
4.1.1.2 test_driveAVX()	24
4.1.1.3 test_driveAVX512()	24
4.1.1.4 test_driveScalar()	24
4.1.1.5 test_driveSSE()	25
4.2 LinCodeWeightInv.h File Reference	25
4.2.1 Detailed Description	27
4.2.2 Function Documentation	27
4.2.2.1 calculate_number_of_words_less_than_fixed_w() [1/3]	27
4.2.2.2 calculate_number_of_words_less_than_fixed_w() [2/3]	27
4.2.2.3 calculate_number_of_words_less_than_fixed_w() [3/3]	28
4.2.2.4 calculate_number_of_words_with_fixed_w() [1/3]	28
4.2.2.5 calculate_number_of_words_with_fixed_w() [2/3]	28
4.2.2.6 calculate_number_of_words_with_fixed_w() [3/3]	29
4.2.2.7 calculateWeightDistribution() [1/3]	29
4.2.2.8 calculateWeightDistribution() [2/3]	29
4.2.2.9 calculateWeightDistribution() [3/3]	30
4.2.2.10 detect()	31
4.2.2.11 find_word_equal_to_fixed_weight() [1/3]	31
4.2.2.12 find_word_equal_to_fixed_weight() [2/3]	31

4.2.2.13 find_word_equal_to_fixed_weight() [3/3]	32
4.2.2.14 find_word_less_than_fixed_weight() [1/3]	32
4.2.2.15 find_word_less_than_fixed_weight() [2/3]	32
4.2.2.16 find_word_less_than_fixed_weight() [3/3]	32
4.2.2.17 min_dis() [1/3]	33
4.2.2.18 min_dis() [2/3]	33
4.2.2.19 min_dis() [3/3]	33
4.2.3 Variable Documentation	34
4.2.3.1 instructionSet	34
4.2.3.2 weights	34
4.3 Data.h File Reference	34
4.3.1 Variable Documentation	35
4.3.1.1 test	35
4.4 ReadWrite.h File Reference	35
4.4.1 Function Documentation	36
4.4.1.1 printMatrix()	36
4.4.1.2 printWeights()	36
4.4.1.3 randomgenf() [1/2]	37
4.4.1.4 randomgenf() [2/2]	37
4.4.1.5 readMatrix()	37
4.4.1.6 write_multpl()	38
4.4.1.7 write_multpl_magma()	38
4.5 Polynomials.h File Reference	39
4.6 lib128.h File Reference	39
4.6.1 Detailed Description	43
4.6.2 Function Documentation	43
4.6.2.1 popcnt_detect()	43
4.6.2.2 popcount()	43
4.6.3 Variable Documentation	44
4.6.3.1 K_FIX	44
4.6.3.2 N_FIX	44
4.6.3.3 POPCNT	44
4.7 lib256.h File Reference	45
4.7.1 Function Documentation	48
4.7.1.1 calculateWeightBytes_256()	48
4.8 lib512.h File Reference	48
4.9 lib_neon.h File Reference	52
4.9.1 Detailed Description	56
4.9.2 Variable Documentation	56
4.9.2.1 K_FIX	56
4.9.2.2 N_FIX	56

Chapter 1

Introduction

1.1 Introduction

The current manual presents a library for calculating some weight characteristics of a linear code. A linear $[n, k]_q$ code is a k -dimensional subspace of the n -dimensional vector space over the Galois field with q elements. The parameters n and k are called length and dimension of C , respectively, and the vectors in C are called codewords. For an arbitrary codeword $v \in C$, its (Hamming) weight $wt(v)$ is defined as the number of its non-zero coordinates. If A_i is the number of codewords of weight i in C , $i = 0, 1, \dots, n$, then the sequence (A_0, A_1, \dots, A_n) is called the weight distribution of C . A $k \times n$ matrix G , whose rows form a basis of C , is called a generator matrix of the code. Our library presents six different functions for calculation of some characteristics of a linear code connected to its weight distribution. The data required for the calculations are: number of field elements (q), code length (n), code dimension (k) and a generator matrix. The main optimizations implemented in the library are based on the *Single Instruction Multiple Data* model of parallel programming. The extended vector registers in the modern Central Processing Units (CPU) are used for the implementation of this parallelization [1, 2]. The optimizations are based on different implementations of algorithms for vector addition over finite fields with $q \leq 64$ elements.

1.1.1 Field Elements Representation

The elements of a finite Galois field F_q can be considered as the residuals modulo a prime number p for $q = p$. In the case of composite field with $q = p^m$ for a prime p and $m > 1$ the elements can be represented as polynomials of degree at most $m - 1$ with coefficients from the prime field F_p . There is a natural representation of the elements of any Galois field as a serial number (or *index*) that allows easy indexation in a computer memory as an integer. The residuals modulo p for a prime p form a prime field with p elements. In the case of composite field the index of each element is calculated by the formula

$$index = \sum_{i=0}^{m-1} a_i p^i, \quad (1.1)$$

where a_i is the coefficient in the polynomial before x^i . The elements of the field can also be presented as the **0** and powers of a primitive element of the field [3]. Therefore, we have three different representations of the elements of a composite field - additive, multiplicative and as a serial number (*index*). The additive representation saves the coefficients of the polynomial as a m -dimensional vector over the prime field. This representation is used for the core computation and optimizations. The multiplicative representations consists of the power of the primitive element. The indexation of the elements of the field shown in formula 1.1 and the multiplicative form are used mostly for the input of the generator matrix. Table 1.1 gives an example of the three main methods of representation of the elements of the field F_9 .

Some of the initialization in the library execute addition and multiplication of the elements of a composite field using their polynomial form. Thus, we have initialized primitive polynomials for the different composite fields. They are initialized in the file *Polynomials.cpp* function *void maketablecomp(int q)*. Table 1.2 shows which primitive irreducible polynomials are used in the current version for the library and some alternative polynomials for the different fields.

Table 1.1 Elements representations for F_9

Index	Polynomial form	Multiplicative form
0	0	9
1	1	x^0
2	2	x^4
3	x	x^1
4	x+1	x^7
5	x+2	x^6
6	2x	x^5
7	2x+1	x^2
8	2x+2	x^3

Table 1.2 Primitive polynomials for $q \leq 64$

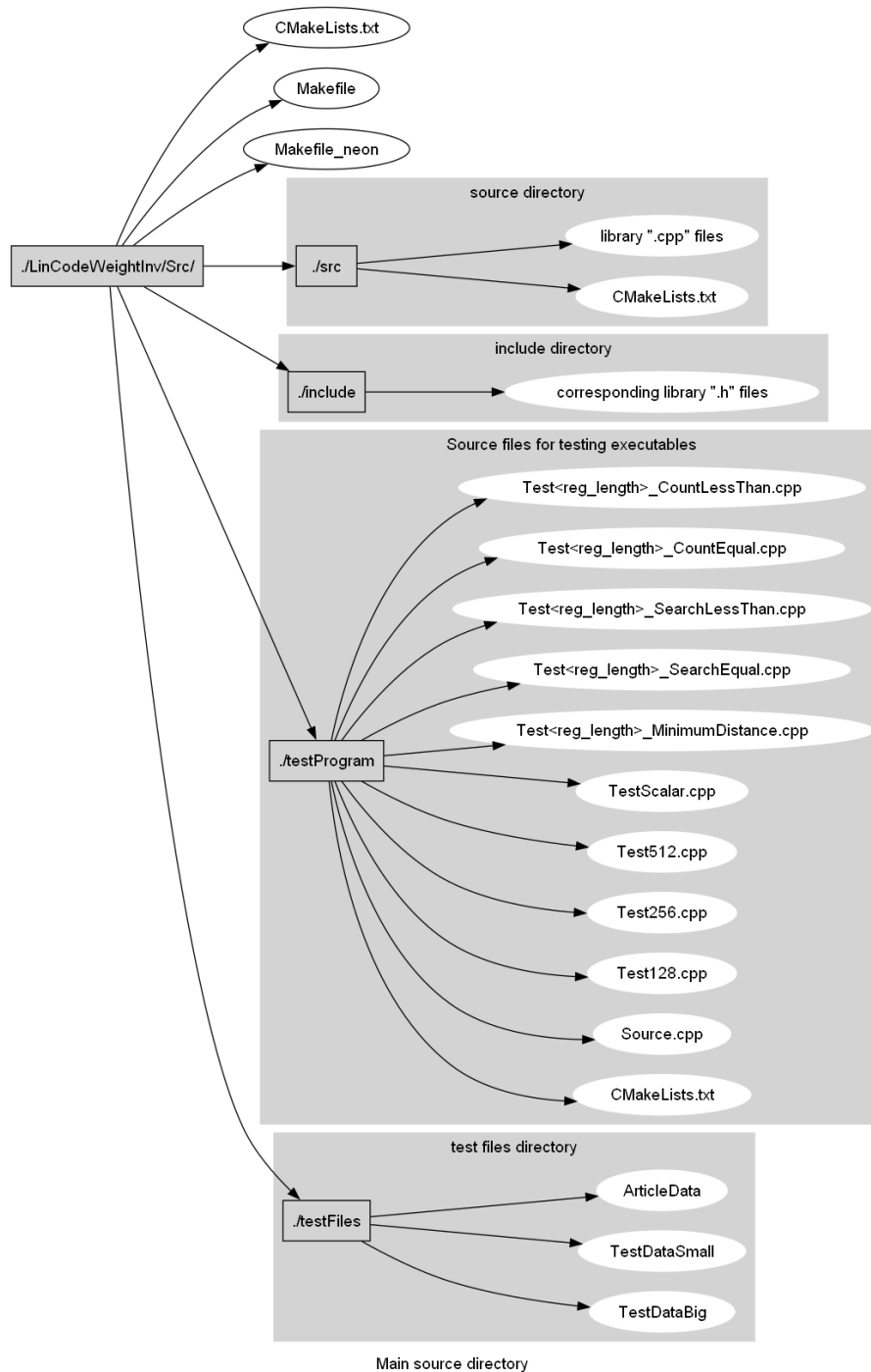
Finite Field	Initialized Primitive Polynomial	Alternative Primitive Polynomials
F_4	$x^2 + x + 1$	-
F_8	$x^3 + x + 1$	$x^3 + x^2 + 1$
F_9	$x^2 + x + 2$	$x^2 + 2x + 2$
F_{16}	$x^4 + x + 1$	$x^4 + x^3 + 1$
F_{25}	$x^2 + x + 2$	$x^2 + 2x + 3; x^2 + 3x + 3; x^2 + 4x + 2$
F_{27}	$x^3 + 2x + 1$	$x^3 + 2x^2 + 1; x^3 + x^2 + 2x + 1; x^3 + 2x^2 + x + 1$
F_{32}	$x^5 + x^2 + 1$	$x^5 + x^3 + 1; x^5 + x^3 + x^2 + x + 1$ $x^5 + x^4 + x^2 + x + 1; x^5 + x^4 + x^3 + x + 1$ $x^5 + x^4 + x^3 + x^2 + 1$
F_{49}	$x^2 + x + 3$	$x^2 + 2x + 3; x^2 + 2x + 5; x^2 + 3x + 5$ $x^2 + 4x + 5; x^2 + 5x + 3; x^2 + 5x + 5; x^2 + 6x + 3$
F_{64}	$x^6 + x + 1$	$x^6 + x^5 + 1; x^6 + x^4 + x^3 + x + 1$ $x^6 + x^5 + x^2 + x + 1; x^6 + x^5 + x^3 + x^2 + 1;$ $x^6 + x^5 + x^4 + x + 1$

Chapter 2

Compilation and Installation

The library source files are contained in the subdirectory *Src* referred as *main directory* in the user manual. Figure 2.1 shows its structure. The current package contains the library, a simple interface program that can be used for calculations and four testing programs that can be used to reproduce the results presented in the manuscript (for 128-, 256- and 512-bit registers and using scalar computations). In this section we present how to use the given Cmake and make files to compile the library and testing programs. For this purpose, a supported C/C++ compiler and *make* or *Cmake* should be installed. If using Cmake the minimum required version is 3.16 or newer. The library is developed for x86 and arm architectures that have NEON instruction set (ARMv8 64-bit architecture or newer). Compilation targeting other architectures is not supported.

Figure 2.1 LinCodeWeightInv directory



2.1 Compilation

2.1.1 Setting up IDE project with CMake

Compiling the library can be done using the CMake framework, which creates a project for the chosen IDE and the working operating system. To do this, the program must be installed, available here. CMake is a platform that, based on a given structure and parameters, creates a project for a selected IDE and the working operating system.

These parameters are set in CMakeLists.txt files created for the current library and application program. For this purpose, the following three CMakeLists.txt files have been created:

- A main file containing the name of the project, the minimum version of CMake required to create the project, and the commands showing the subdirectories where the rest of the *CMakeLists.txt* files are located. This file is located in the root directory.
- A file describing the structure and how to compile the library itself. This file is located in the **src** subdirectory, which also contains all the necessary source files to build the current library. The **include** subdirectory of the main directory contains the library's header files.
- A file describing the structure of the attached sample program, testing programs with different instruction sets and linking to the library. This file and the source files are located in the **testProgram** subdirectory of the main directory.

There are two main approaches to configure a IDE project - using the Graphics User Interface (GUI) or by a command in a standard terminal (typically in Unix based systems). The two main steps for configuration using the GUI are shown in fig. 2.2 and 2.3.

Figure 2.2 CMake configuration step

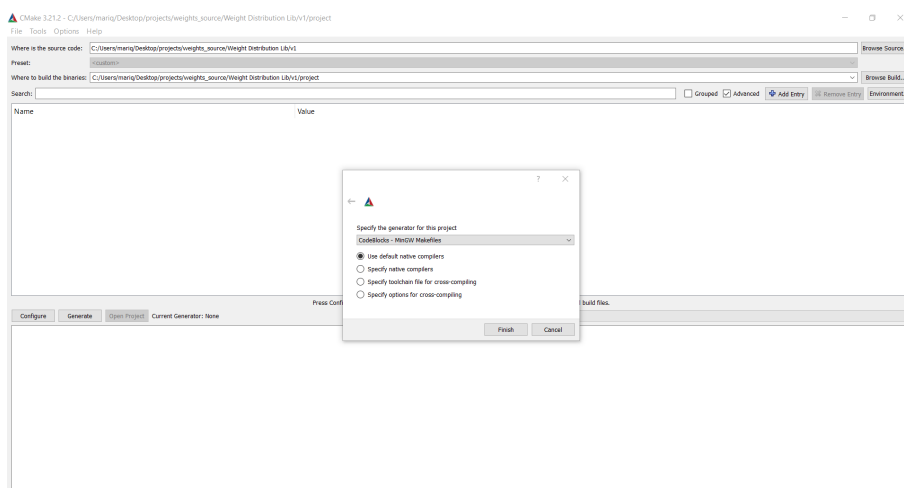
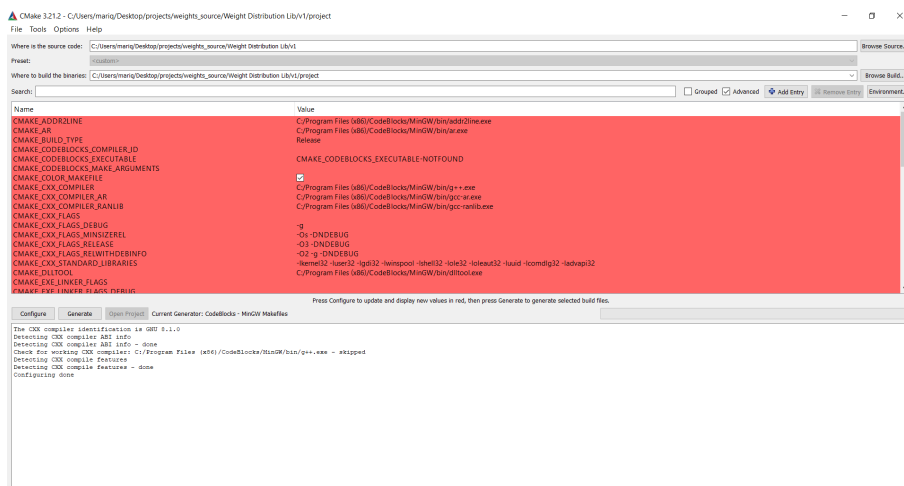


Figure 2.3 CMake generation step



The following command can be used to configure a project for an IDE using command prompt from the main directory of the library:

```
cmake -G <build_system> -B <IDE_project_dir> -S <source_dir>
```

Example 1: Configuring and generating IDE project on Windows system for Visual Studio 16 and CodeBlocks in a subfolder "msvc"

```
cmake -G "Visual Studio 16 2019" -B .\ msvc -S .\
cmake -G "CodeBlocks" -B .\ msvc -S .\
```

Example 2: Configuring and generating IDE project on MacOS for XCode in subdirectory "xcode"

```
cmake -G XCode -B ./xcode -S ./
```

Note: The command `cmake --help` will produce additional information and the available generators for the current platform.

Note: The command `cmake -D CMAKE_C_COMPILER=<full_compiler_path>` can be used before the configuration command to set a specific C compiler. Analogously, the command `cmake -D CMAKE_CXX_COMPILER=<full_compiler_path>` can be used to set a specific C++ compiler. However, when working with an IDE such as Microsoft Visual Studio the authors recommend to use the default compiler.

Example 3: Setting up a specific compiler using its name (compiler path is set in **PATH**) using default generator on Unix based Operating system:

```
cmake -D CMAKE_C_COMPILER=gcc -D CMAKE_CXX_COMPILER=g++ -B ./ buildDir -S ./
```

2.1.2 Compiling with IDE

In the generated project with Cmake and the chosen IDE, there are several targets that can be build (compiled). Building target **ALL_BUILD** (it can be named differently in the chosen IDE e.g. in CodeBlocks - target **all**) will compile all executable programs. One can build a specific executable program by right-clicking on it and choosing option *build*. However, they will not be able to be executed directly from the IDE. Thus, they need to be set as *default starting program*. Figure 2.4 shows how to set a default starting program for a xcode on macOS. Figure 2.5 shows how to chose a target to be build as default starting program for Microsoft Visual Studio and figure 2.6 shows this for CodeBlocks IDE both working on Windows 10 OS.

2.1.3 Compiling on Linux OS

To compile the library and the accompanying programs on a Unix OS with default generator set as *Unix Makefiles*, the following command must be executed:

```
cmake -B ./ -S ./
make all
```

The Sample interface program and the testing programs are compiled in subdirectory *build* of the chosen IDE directory. The following commands can be executed to start the Sample program:

```
cd build
./Test-UI
```

Figure 2.4 Setting test program as startup project (XCode)

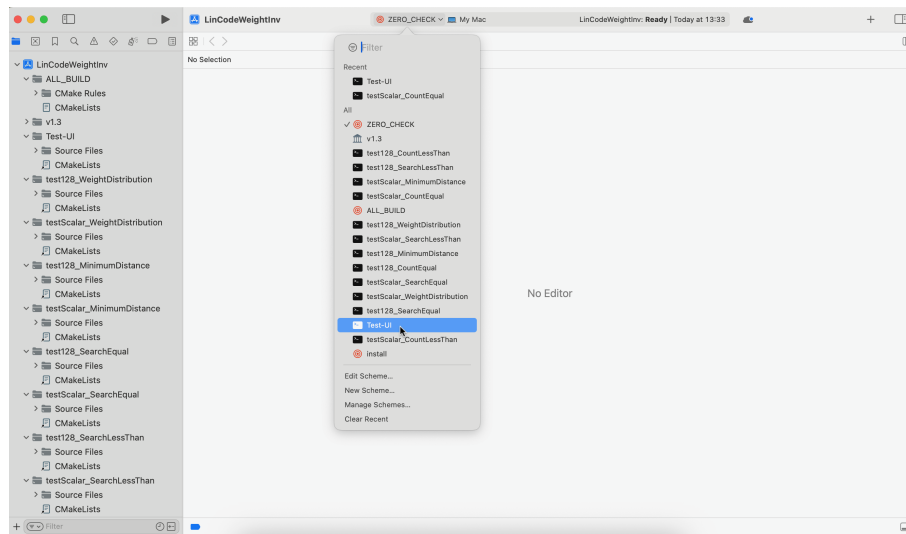


Figure 2.5 Setting test program as startup project (Microsoft Visual Studio)

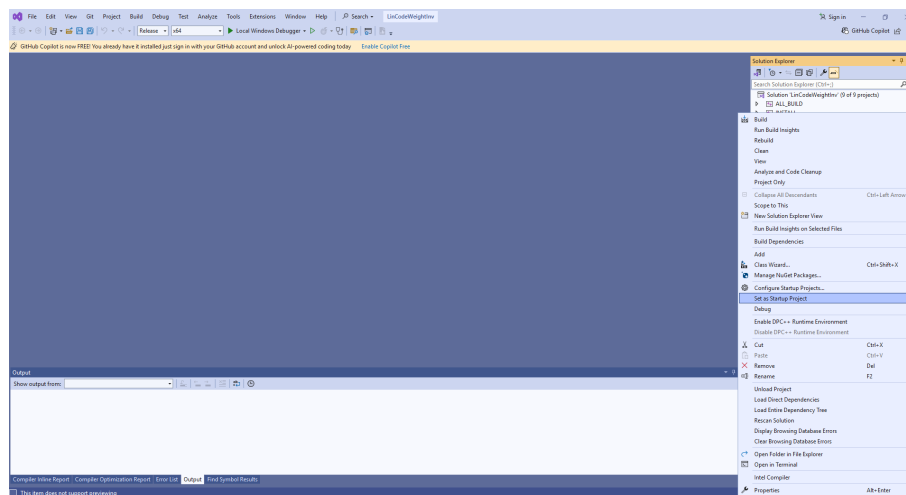
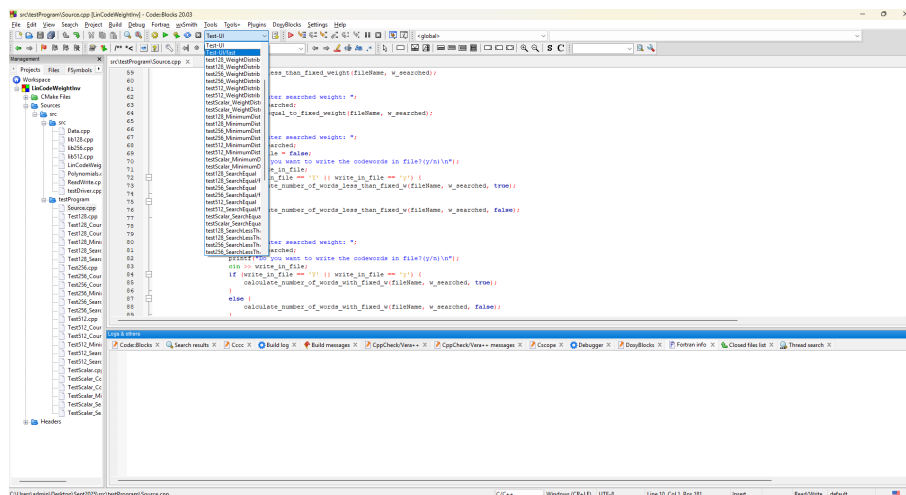


Figure 2.6 Setting test program as startup project (CodeBlocks)



2.1.4 Compiling for AVX512

When using Microsoft Visual Studio IDE on Windows OS, the user should manually set the flag to *AVX512* in the project or change it in the corresponding CMakeLists file as can be seen in the following example. The default flag compiles the library for architectures with AVX2 instruction sets since they are more widely available. If the target platform has **AVX512-VPOPCNTDQ** instruction set the flag *-mavx512vpopcntdq* (in comment on line 23) should be added to the flags set on line 20 in *CMakeLists.txt* file in subdirectory *src*. Otherwise, these instructions will not be used.

Example 4: Flags set in *CMakeLists.txt* file in subdirectory *src* for use with **AVX512-VPOPCNTDQ**.

```
if(CMAKE_CXX_COMPILER_ID STREQUAL "MSVC")
set(CMAKE_CXX_FLAGS "$CMAKE_CXX_FLAGS /arch:AVX512")
else()
set(CMAKE_CXX_FLAGS "$CMAKE_CXX_FLAGS -march=native -mavx512vpopcntdq")
endif()
```

2.1.5 Compilation using the included Makefiles

On a Unix-like x86 systems, the included *Makefile* can be used to compile the library and the presented Sample program. If the system architecture is *ARM* with NEON instruction set, the file *Makefile_neon* should be renamed to *Makefile* (override the existing Makefile). Then the following commands will compile and install the library (saved in *lib* subdirectory, test files will be copied in *build* subdirectory) and the executable Sample program (in subdirectory *build*):

```
make
make install
```

2.1.6 Compilation using specific instruction set on x86 architecture

The end user can use a specific instruction set (SSE4.1, AVX2, AVX512), different from the largest possible on the target architecture. To do this the end user should change the value of **FORCE_INSTR** parameter in the *CMakeList.txt* file in subdirectory *src* to one of the following:

- For SSE4.1 change the value to 5
- For AVX2 change the value to 8
- For AVX512 without additional population count instruction set change the value to 9
- For AVX512 with additional population count instruction set change the value to 11
- Default value of **FORCE_INSTR** is 0 and it indicates to use the largest register available instruction set

This should be done before the generation of the project.

2.2 Installation

The created project has the following main targets - the library, the accompanying interface program, four testing programs used for reproducing the results presented in the manuscript and the install target. Setting the test target as default will compile the library and all executable programs. The interface program will be the one that is started in the corresponding IDE. Building the install target will generate a library in the subdirectory *lib* of the main directory, while the compiled sample program is in subdirectory *build* of the chosen directory for the generator (IDE, Makefile, etc.). It will also copy the test files *TestDataSmall* and *TestDataBig* into the build directory. On an Unix system with default generator set as *Unix Makefiles* the library and the accompanying Sample program can be configured, build and installed (Makefile stored in subdirectory *make*) with the following set of commands:

```
cmake -B ./ make -S ./  
cd make  
make all  
make install
```

2.2.1 Using library in personal projects

There are two main ways to use the library in personal projects:

- By using the *CMakeLists.txt* file for the sample program - replacing the source *.cpp* file for the sample program with personal source *.cpp* files. This negates the need to manually enter the directories for the compilation and linking processes.
- By adding the appropriate directories for the library in the personal project in the preferred IDE. The *include* directory contains the needed header files for the compilation and the *lib* directory containing the library for the linking process.

Note: The library must be compiled in the same mode and with the same compiler that will be used for the personal projects.

2.3 Tested platforms

The *LinCodeWeoghtInv* library has been tested on the platforms, given in table 2.1:

Table 2.1 Tested platforms

CPU	OS	Instruction set	Compiler	Build system or IDE
Intel Xeon Gold 5118 @2.36GHz	Red Hat Enterprise 7.9	AVX512 F/BW	gcc 9.2	Makefile
Intel Core i5-13600K @3.5GHz	Windows 11 Pro	AVX2	gcc 10.3 msvc 19.37.32724	CodeBlocks 20.03 Visual Studio Community 17
Intel Core i5-1035G @1.0GHz	Windows 10 Pro	AVX512 F/BW/ VPOPCNTDQ	gcc 8.1 msvc 19.29.30146	CodeBlocs 20.03 Visual Studio Community 16
Intel Xeon E5-2640 @2.5GHz	Ubuntu 18.04	SSE4.2	gcc 7.5	Makefile CodeBlocks 20.03
AMD Ryzen 5 7600 @3.8GHz	Windows 11 Home	AVX512 F/BW/ VPOPCNTDQ	gcc 8.1 msvc 19.39.33523	CodeBlocks 20.03 Visual Studio Community 17
Apple M1 @3.2 GHz	macOS Sonoma 14.1.2	NEON	clang 15.0.0	Xcode 15.0.2
Cortex-A76 (emulated with qemu)	Ubuntu Server 24.04 (arm64)	NEON	clang 18.1.3	Makefile

Chapter 3

Sample and testing executable programs

The software package includes a Sample program with simple interface and a set of executable programs that execute a single function for a specific instruction set. All of the including programs must be firstly compiled (see Chapter 2).

3.1 Sample program

The Sample program with source code in subdirectory **testProgram**. It presents a simple interface that allows the user to test the main functionalities. The first option gives the end user to execute calculations with input data that is stored in a file. The name of the file is read from the command prompt. Afterwards one of the six main functionalities of the library, described in 4.2:

- Calculating the weight distribution of a linear code.
- Calculating the minimum distance of the code.
- Searching for codeword with weight less than given value. The value is read from the command prompt.
- Searching for codeword with weight equal to given value. The value is read from the command prompt.
- Calculating the number of codewords with weight less than given value. The value is read from the command prompt.
- Calculating the number of codewords with weight equal to given value. The value is read from the command prompt.

Figures 3.1,3.2and 3.3 show the structure of the data in the input file. An error occurs if the input data (generator matrix, code parameters) are not in the appropriate format in the input file. A row that starts with "?" shows that the element of the field are represented using the index of the element of the field (decimal representation). Afterwards, the parameters k , n and q are given. The last number is the current code for which the calculations are executed. The generator matrix is given in the next rows. If a row starts with "!", then the elements of the field are displayed in multiplicative form.

Figure 3.1 File structure for $q < 10$ and decimal representation of the elements of the field

```

1
2 ? 3 10 2 1
3 1001010101
4 0101110000
5 0011110100
6

```

Normal text file length: 51 lines: 6 Ln: 2 Col: 2 Pos: 4 Windows (CR LF) UTF-8 INS

Figure 3.2 File structure for $q > 10$ and decimal representation of the elements of the field

```

1
2 ? 3 10 25 1
3 1,0,0,1,19,9,17,15,13,14,
4 0,1,0,1,23,13,2,24,3,4,
5 0,0,1,1,18,5,23,23,7,2,
6

```

Normal text file length: 93 lines: 6 Ln: 2 Col: 2 Pos: 4 Windows (CR LF) UTF-8 INS

Figure 3.3 File structure for $q > 10$ and multiplicative representation of the elements of the field

```

1
2 ! 3 10 16 1
3 0,16,16,0,6,2,10,3,5,10,
4 16,0,16,0,1,14,11,5,10,10,
5 16,16,0,0,0,4,16,11,11,4,
6

```

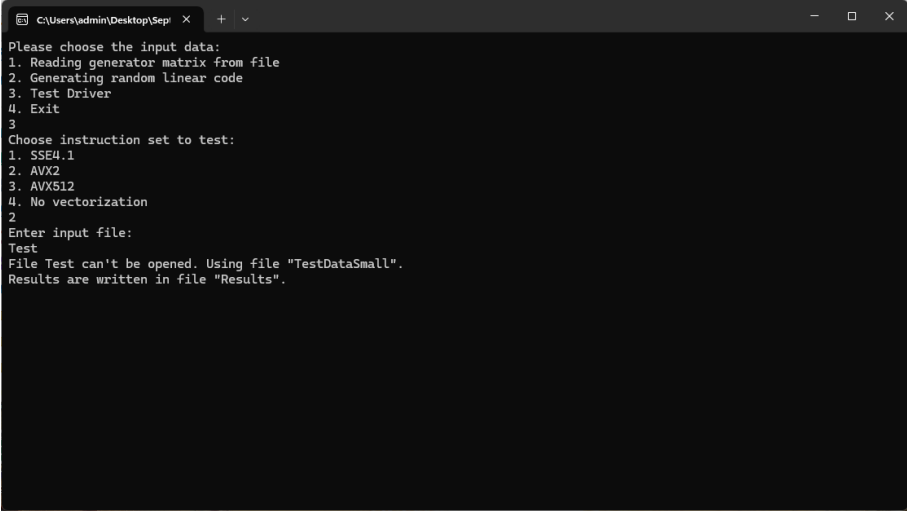
Normal text file length: 97 lines: 6 Ln: 2 Col: 2 Pos: 4 Windows (CR LF) UTF-8 INS

The second option is to run calculation for a randomly generated linear code. The values for the parameters of the code n , k and q are read from the command prompt. Then the user chooses which calculations from the above to be executed.

The last option of the presented sample program is to execute the so called "testDriver" function. It allows for the testing of all functions of the LinCodeWeightInv library for correctness and runtime testing. The "testDriver" consists of two parts - data generation and testing functionality. The first is preliminary and it is not executed by calling the "testDriver" function. It concerns the generation of test data. We have created several files that contain codes with randomly generated parameters over each of the considered finite fields, and 100 randomly generated codes for each of the parameters. In addition to the generator matrices of the codes, the files also contain pre-calculated weight distributions with other software that uses a Gray code. This procedure (of generator matrices and weight distributions) is repeated several times (say 3). We have created two such files (*Test* and *TestDataRes_big*) according to the maximum number of codewords in each of the codes (*Test* contains codes with smaller dimensions than *TestDataRes_big* – the reason is to have test data that can be checked faster). These files have the following structure: parameters of the code, presented as in Figures 3.1 and 3.2, generator matrix and weight distribution $\{A_i\}$ for $i = 0, \dots, n$. For a given set of parameters we have a 100 codes presented by their generator matrices with computed weight distributions.

The Sample program also includes a test drive option. It reads a generator matrix and a weight distribution from the previously prepared files, and then tests the correctness of each of the functions and the execution time (respectively, the average time for the hundred codes with the same parameters), saving the results to a separate file. We only test the correctness of the calculations and the execution time. The interface allows the user to choose with which instruction set to test the library - SSE4.1, AVX2, AVX512 or without the use of vectorization. If an ARM architecture with NEON instruction set is detected, an appropriate message is displayed. Figure 3.4 shows the console look of Sample program with the input using "Test Driver" option.

Figure 3.4 Example of console input in the Sample program using "Test Driver" option



```

C:\Users\admin\Desktop\Sep  X  +  -  x
Please choose the input data:
1. Reading generator matrix from file
2. Generating random linear code
3. Test Driver
4. Exit
3
Choose instruction set to test:
1. SSE4.1
2. AVX2
3. AVX512
4. No vectorization
2
Enter input file:
Test
File Test can't be opened. Using file "TestDataSmall".
Results are written in file "Results".

```

The following is the source code of the Sample program:

```

#include <time.h>
#include <iostream>
#include <fstream>
#include <stdio.h>
#include <string.h>
#include "testDriver.h"
#include "LinCodeWeightInv.h"
using namespace std;
int main(int argc, char** argv) {
    int flag = 0;
    int k = 3 , n = 500, q = 25, w_searched = 480, count = 0;

```

```
unsigned long long int d = 0;
bool found = false, multiplicative = false, write_CW = true;

char write_in_file = false;
while (flag!=4) {
    printf("Please choose the input data:\n");
    printf("1. Reading generator matrix from file\n");
    printf("2. Generating random linear code\n");
    printf("3. Test functionalities\n");
    printf("4. Exit\n");
    cin >> flag;
    if (flag == 1) {
        cout << "Enter file name: ";
        string in = "";
        char* fileName;
        cin >> in;
        fileName = new char[in.length() + 1];
        strcpy(fileName, in.c_str());
        int num_of_input = 0;
        int function = 0;
        while (function < 7) {
            printf("Please choose an option:\n");
            printf("1 Calculating weight spectrum\n");
            printf("2 Calculating minimum distance of code\n");
            printf("3 Searching for word with weight less than:\n");
            printf("4 Searching for word with weight equal to:\n");
            printf("5 Calculating number of words with weight less than:\n");
            printf("6 Calculating number of words with weight equal to:\n");
            printf("7 Back\n");
            cin >> function;
            switch (function) {
                case 1:
                    calculateWeightDistribution(fileName);
                    break;
                case 2:
                    min_dis(fileName);
                    break;
                case 3:
                    cout << "Enter searched weight: ";
                    cin >> w_searched;
                    find_word_less_than_fixed_weight(fileName, w_searched);
                    break;
                case 4:
                    cout << "Enter searched weight: ";
                    cin >> w_searched;
                    find_word_equal_to_fixed_weight(fileName, w_searched);
                    break;
                case 5:
                    cout << "Enter searched weight: ";
                    cin >> w_searched;
                    write_in_file = false;
                    printf("Do you want to write the codewords in file?(y/n)\n");
                    cin >> write_in_file;
                    if (write_in_file == 'Y' || write_in_file == 'y') {
                        calculate_number_of_words_less_than_fixed_w(fileName, w_searched, true);
                    }
                    else {
                        calculate_number_of_words_less_than_fixed_w(fileName, w_searched, false);
                    }
                    break;
                case 6:
                    cout << "Enter searched weight: ";
                    cin >> w_searched;
```

```

printf("Do you want to write the codewords in file?(y/n)\n");
cin >> write_in_file;
if (write_in_file == 'Y' || write_in_file == 'y') {
    calculate_number_of_words_with_fixed_w(fileName, w_searched, true);
}
else {
    calculate_number_of_words_with_fixed_w(fileName, w_searched, false);
}
break;
case 7:
break;
default:
printf("Wrong input\n");
break;
}
}
}
else if (flag == 2) {
cout << "Enter values for n,k,q" << endl;
cin >> n >> k >> q;
int function = 0;
printf("Please choose an option:\n");
printf("1 Calculating weight spectrum\n");
printf("2 Calculating minimum distance of code\n");
printf("3 Searching for word with weight less than:\n");
printf("4 Searching for word with weight equal to:\n");
printf("5 Calculating number of words with weight less than:\n");
printf("6 Calculating number of words with weight equal to:\n");
cin >> function;
switch (function) {
case 1:
calculateWeightDistribution(n, k, q);
for (int i = 1; i <= n; i++) {
    if (weights[i] > 0) {printf("%d^%llu ", i, weights[i]);}
}
printf("\n");
break;
case 2:
d = min_dis(n, k, q);
printf("d = %d\n", d);
break;
case 3:
cout << "Enter searched weight: ";
cin >> w_searched;
found = find_word_less_than_fixed_weight(n, k, q, w_searched);
if (found) {printf("Found a word with weight less than %d\n", w_searched);}
else {printf("Did NOT find a word with weight less than %d\n", w_searched);}
break;
case 4:
cout << "Enter searched weight: ";
cin >> w_searched;
found = find_word_equal_to_fixed_weight(n, k, q, w_searched);
if (found) {printf("Found a word with weight = %d\n", w_searched);}
else {printf("Did NOT find a word with weight = %d\n", w_searched);}
break;
case 5:
cout << "Enter searched weight: ";
cin >> w_searched;
write_in_file = false;
printf("Do you want to write the codewords in file?(y/n)\n");
cin >> write_in_file;
if (write_in_file == 'Y' || write_in_file == 'y') {
    unsigned long long int num =

```

```

    calculate_number_of_words_less_than_fixed_w(n, k, q, w_searched, true);
    printf("Found %d words with weight less than %d\n", num, w_searched);}
else {
    unsigned long long int num =
    calculate_number_of_words_less_than_fixed_w(n, k, q, w_searched, false);
    printf("Found %d words with weight less than %d\n", num, w_searched);}
break;
case 6:
cout << "Enter searched weight: ";
cin >> w_searched;
write_in_file = 'n';
printf("Do you want to write the codewords in file?(y/n)\n");
cin >> write_in_file;
if (write_in_file == 'Y' || write_in_file == 'y') {
    unsigned long long int num =
    calculate_number_of_words_with_fixed_w(n, k, q, w_searched, true);
    printf("Found %d words with weight = %d\n", num, w_searched); }
else {
    unsigned long long int num =
    calculate_number_of_words_with_fixed_w(n, k, q, w_searched, false);
    printf("Found %d words with weight = %d\n", num, w_searched);}
break;
default:
printf("Wrong input\n");
break;}}
else if(flag == 3){test_drive();}
else if (flag == 4) {break; }
else {printf("Not an option\n\nPlease choose again\n");}
printf("\n\n\n"); }
return 0;
}

```

3.2 Testing executable programs

In order to test all functionalities with the exact input data used in the accompanying article, we have developed additional functions and corresponding executable programs. There are 24 programs that can be compiled and executed (see Section 2.1.2). The input data is stored in the file *ArticleData*. Manual input is not needed for the execution of these programs. The input file contains codes for seven finite fields, fixed dimension for each field and lengths 60 and 500. For each set of parameters, there are 100 randomly generated codes and their weight distribution. The output of the testing programs contains the average execution time of the function for each set of 100 codes with the same parameters and the total execution time. The results are written in a file with the name *Results_<FunctionName>_<InstructionSet>*, where *FunctionName* shows the used functions and *<InstructionSet>* gives information about the desired instruction set that is going to be used. The searching and counting functions write the codewords, that were found in a separate file. If an error occurs during the execution, information about it will be written in a text file *error.txt* and the program will end with code 1. For example, when executing the program *test512_WeightDistribution* on an architecture with AVX512 register set and default setting (see 2.1.4), the following message will be written in the *error.txt* file:

AVX512 instructions were detected on CPU, but the corresponding compiler flag was not used! If working on Visual Studio, please change compiler flag at line 18 in CMakeLists.txt file in src directory to /arch:AVX512, or manually change the flags for project v1.3! Otherwise, please change the flag to -march=skylake-avx512 in line 20 in the same in CMakeLists.txt file in src directory.

Table 3.1 gives the names of the executable testing programs, the names of the output files, and total execution time in seconds. The computations are executed on platform with Linux OS, Intel Core i5-1035G CPU and g++ 13.3.0.

Table 3.1 Execution times for functions calculating the weight distribution and searching for codewords with given weight

Testing program	Instruction set	Output file	Total time
test128_WeightDistribution	SSE4.1	Results_WeightDistribution_SSE	2 h. 10 min.
test256_WeightDistribution	AVX2	Results_WeightDistribution_AVX	2 h. 9 min.
test512_WeightDistribution	AVX512	Results_WeightDistribution_AVX512	1 h. 13 min.
testScalar_WeightDistribution	no vectorization	Results_WeightDistribution_Scalar	2 d. 6 h. 37 min.
test128_SearchEqual	SSE4.1	Results_SearchEqual_SSE	5 h. 15 min.
test256_SearchEqual	AVX2	Results_SearchEqual_AVX	5 h. 23 min.
test512_SearchEqual	AVX512	Results_SearchEqual_AVX512	3 h. 2 min.
testScalar_SearchEqual	no vectorization	Results_SearchEqual_Scalar	5 d. 5 h. 57 min.

Chapter 4

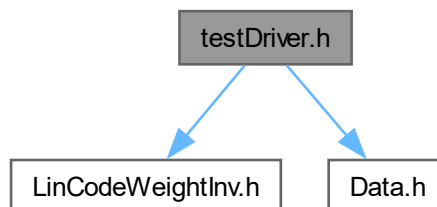
File Documentation

4.1 testDriver.h File Reference

Declaration of a function for correctness and runtime testing.

```
#include "LinCodeWeightInv.h"  
#include "Data.h"
```

Include dependency graph for testDriver.h:



Functions

- void **test_drive** ()
This function gives a simple interface designed to test all end user functionalities of the presented library with different instruction sets.
- void **test_driveSSE** (int mode)
Function for testing functionalities with data from the corresponding manuscript with registers with 128-bit registers.
- void **test_driveAVX** (int mode)
Function for testing functionalities with data from the corresponding manuscript with registers with 256-bit registers.
- void **test_driveAVX512** (int mode)
Function for testing functionalities with data from the corresponding manuscript with registers with 512-bit registers.
- void **test_driveScalar** (int mode)
Function for testing functionalities with data from the corresponding manuscript with registers without vectorization.

4.1.1 Function Documentation

4.1.1.1 test_drive()

```
void test_drive ( )
```

The interface lets the user to chose with which instruction sets the computations will be executed. If ARM architecture is detected, appropriate message is displayed. Then the user should enter the name of a testing file. If it does not exists, it searches for file *Test* in the current directory. The testing file should have the following structure: parameters of a code, presented as in Figures 3.1 and 3.2, generator matrix and weight distribution $\{A_i\}$ for $i = 0, \dots, n$. For a given set of parameters we have a 100 codes presented by their generator matrices with computed weight distributions.

Note

The results (average execution times, total computational time) of the calculations are written in file **Results**. The program ends with error message if there are errors in the computations. Error message is written in file **error.txt**

4.1.1.2 test_driveAVX()

```
void test_driveAVX (
    int mode)
```

Parameters

<i>mode</i>	Shows which function will be tested. Possible values are 1, 2, 3, 4, 5, 6 that correspond to the functions for calculating weight distribution, minimum distance, searching for codeword with weight equal to or less than given value, counting the codewords with weight equal to or less than given value, respectively.
-------------	---

4.1.1.3 test_driveAVX512()

```
void test_driveAVX512 (
    int mode)
```

Parameters

<i>mode</i>	Shows which function will be tested. Possible values are 1, 2, 3, 4, 5, 6 that correspond to the functions for calculating weight distribution, minimum distance, searching for codeword with weight equal to or less than given value, counting the codewords with weight equal to or less than given value, respectively.
-------------	---

4.1.1.4 test_driveScalar()

```
void test_driveScalar (
    int mode)
```


Parameters

<i>mode</i>	Shows which function will be tested. Possible values are 1, 2, 3, 4, 5, 6 that correspond to the functions for calculating weight distribution, minimum distance, searching for codeword with weight equal to or less than given value, counting the codewords with weight equal to or less than given value, respectively.
-------------	---

4.1.1.5 test_driveSSE()

```
void test_driveSSE (
    int mode)
```

Parameters

<i>mode</i>	Shows which function will be tested. Possible values are 1, 2, 3, 4, 5, 6 that correspond to the functions for calculating weight distribution, minimum distance, searching for codeword with weight equal to or less than given value, counting the codewords with weight equal to or less than given value, respectively.
-------------	---

4.2 LinCodeWeightInv.h File Reference

Declaration of end user functions.

Since, only the described bellow functions are to be available to the end user, here we present the dependency graph for the corresponding *.cpp* file. It uses functions from all of the described *.h* files. Figure 4.1 shows the dependency graph for *x86* architectures. Figure 4.2 shows the dependency graph for *ARM* architectures with NEON instruction sets.

Figure 4.1 Dependency graph for *LinCodeWweightInv.cpp* for *x86* architectures

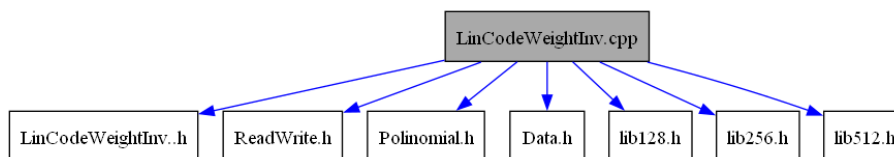
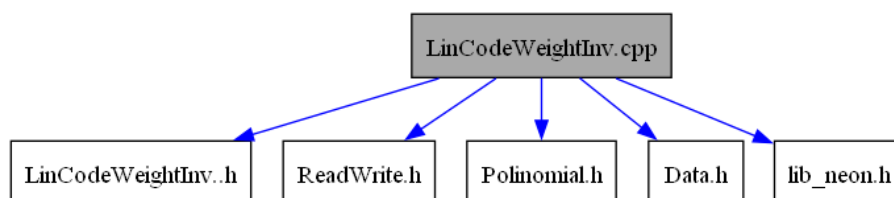


Figure 4.2 Dependency graph for *LinCodeWweightInv.cpp* for *ARM* architectures with NEON instruction sets



Functions

- void **detect** ()
Global function that detects the available instruction set on the specific hardware.
- void **calculateWeightDistribution** (int **generatorMatrix, int N, int K, int Q, bool multiplicativeForm)
Calculation of the weight spectrum of a linear code for a given generator matrix.
- void **calculateWeightDistribution** (int N, int K, int Q)
Calculation of the weight spectrum of a randomly generated linear code for given parameters of the code.
- void **calculateWeightDistribution** (char *generatorMatrixFile)
Calculating the weight distribution of linear codes for given generator matrices, written in a file.
- unsigned long long int **min_dis** (int **generatorMatrix, int N, int K, int Q, bool multiplicativeForm)
Calculating the minimum distance of a linear code for a given generator matrix.
- unsigned long long int **min_dis** (int N, int K, int Q)
Calculating the minimum distance of a randomly generated linear code for given parameters of the code.
- void **min_dis** (char *generatorMatrixFile)
Calculating the minimum distance of linear codes for given generator matrices, written in a file.
- bool **find_word_less_than_fixed_weight** (int **generatorMatrix, int N, int K, int Q, int W, bool multiplicativeForm)
Searching for a codeword with weight less than a given value of a linear code for a given generator matrix.
- bool **find_word_less_than_fixed_weight** (int N, int K, int Q, int W)
Searching for a codeword with weight less than a given value of a randomly generated linear code for given parameters of the code.
- void **find_word_less_than_fixed_weight** (char *generatorMatrixFile, unsigned long long int W)
Searching for a codeword with weight less than a given value for given generator matrices, written in a file.
- bool **find_word_equal_to_fixed_weight** (int **generatorMatrix, int N, int K, int Q, int W, bool multiplicativeForm)
Searching for a codeword with weight equal to a given value of a linear code for a given generator matrix.
- bool **find_word_equal_to_fixed_weight** (int N, int K, int Q, int W)
Searching for a codeword with weight equal to a given value of a randomly generated linear code for given parameters of the code.
- void **find_word_equal_to_fixed_weight** (char *generatorMatrixFile, unsigned long long int w_searched)
Searching for a codeword with weight equal to a given value for given generator matrices, written in a file.
- unsigned long long int **calculate_number_of_words_less_than_fixed_w** (int **generatorMatrix, int N, int K, int Q, int w, bool write, bool multiplicativeForm)
Calculating the number of codewords with weight less than a given value for a given generator matrix.
- unsigned long long int **calculate_number_of_words_less_than_fixed_w** (int N, int K, int Q, int w, bool write)
Calculating the number of codewords with weight less than a given value of a randomly generated linear code for given parameters of the code.
- void **calculate_number_of_words_less_than_fixed_w** (char *generatorMatrixFile, unsigned long long int w_searched, bool write)
Calculating the number of codewords with weight less than a given value for given generator matrices, written in a file.
- unsigned long long int **calculate_number_of_words_with_fixed_w** (int **generatorMatrix, int N, int K, int Q, int w, bool write, bool multiplicativeForm)
Calculate number of words with weight equal to a variable for a given generator matrix.
- unsigned long long int **calculate_number_of_words_with_fixed_w** (int N, int K, int Q, int w, bool write)
Calculate number of words with weight equal to a variable of a randomly generated linear code for given parameters of the code.
- void **calculate_number_of_words_with_fixed_w** (char *generatorMatrixFile, unsigned long long int w_searched, bool write)
Calculate number of words with weight equal to a variable for given generator matrices, written in a file.

Variables

- unsigned long long int **weights** []
Global array, containing the (partial) weight distribution of a code, after calculations.
- int **instructionSet**
Global variable that shows the instruction set that is used.

4.2.1 Detailed Description

Declaration of structure and functions used in computations in composite fields.

4.2.2 Function Documentation

4.2.2.1 calculate_number_of_words_less_than_fixed_w() [1/3]

```
void calculate_number_of_words_less_than_fixed_w (
    char * generatorMatrixFile,
    unsigned long long int w_searched,
    bool write )
```

Parameters

<i>generatorMatrixFile</i>	Name of the file, containing the generator matrices
<i>W</i>	value of the searched weight of a codeword
<i>write</i>	if true, the calculated nonproportional codewords will be written in file "Result_codewords.txt"

Note

The result for each linear code from the input is written in an output file "count_less_than.txt"

4.2.2.2 calculate_number_of_words_less_than_fixed_w() [2/3]

```
unsigned long long int calculate_number_of_words_less_than_fixed_w (
    int ** generatorMatrix,
    int N,
    int K,
    int Q,
    int w,
    bool write,
    bool multiplicativeForm )
```

Parameters

<i>generatorMatrix</i>	Generator matrix, saved in 2-dimensional array
<i>N</i>	Length of the code
<i>K</i>	Dimension of the code
<i>Q</i>	Number of elements in the finite field
<i>W</i>	value of the searched weight of a codeword
<i>write</i>	if true, the calculated nonproportional codewords will be written in file "Result_codewords.txt"
<i>multiplicativeForm</i>	true if the elements of the field are written in multiplicative form

4.2.2.3 calculate_number_of_words_less_than_fixed_w() [3/3]

```
unsigned long long int calculate_number_of_words_less_than_fixed_w (
    int N,
    int K,
    int Q,
    int w,
    bool write )
```

Parameters

<i>N</i>	Length of the code
<i>K</i>	Dimension of the code
<i>Q</i>	Number of elements in the finite field
<i>W</i>	value of the searched weight of a codeword
<i>write</i>	if true, the calculated nonproportional codewords will be written in file "Result_codewords.txt"

4.2.2.4 calculate_number_of_words_with_fixed_w() [1/3]

```
void calculate_number_of_words_with_fixed_w (
    char * generatorMatrixFile,
    unsigned long long int w_searched,
    bool write )
```

Parameters

<i>generatorMatrixFile</i>	Name of the file, containing the generator matrices
<i>W</i>	value of the searched weight of a codeword
<i>write</i>	if true, the calculated nonproportional codewords will be written in file "Result_codewords.txt"

Note

The result for each linear code from the input is written in an output file "count_equal.txt"

4.2.2.5 calculate_number_of_words_with_fixed_w() [2/3]

```
unsigned long long int calculate_number_of_words_with_fixed_w (
    int ** generatorMatrix,
    int N,
    int K,
    int Q,
    int w,
    bool write,
    bool multiplicativeForm )
```

Parameters

<i>generatorMatrix</i>	Generator matrix, saved in 2-dimensional array
<i>N</i>	Length of the code
<i>K</i>	Dimension of the code
<i>Q</i>	Number of elements in the finite field
<i>W</i>	value of the searched weight of a codeword
<i>write</i>	if true, the calculated nonproportional codewords will be written in file "Result_codewords.txt"
<i>multiplicativeForm</i>	true if the elements of the field are written in multiplicative form

4.2.2.6 calculate_number_of_words_with_fixed_w() [3/3]

```

unsigned long long int calculate_number_of_words_with_fixed_w (
    int N,
    int K,
    int Q,
    int w,
    bool write )

```

Parameters

<i>N</i>	Length of the code
<i>K</i>	Dimension of the code
<i>Q</i>	Number of elements in the finite field
<i>W</i>	value of the searched weight of a codeword
<i>write</i>	if true, the calculated nonproportional codewords will be written in file "Result_codewords.txt"

4.2.2.7 calculateWeightDistribution() [1/3]

```

void calculateWeightDistribution (
    char * generatorMatrixFile )

```

Parameters

<i>generatorMatrixFile</i>	Name of the file, containing the generator matrices
----------------------------	---

Note

The weight distribution is written in output file "Weight_dis.txt"

4.2.2.8 calculateWeightDistribution() [2/3]

```

void calculateWeightDistribution (
    int ** generatorMatrix,
    int N,
    int K,
    int Q,
    bool multiplicativeForm )

```

Parameters

<i>generatorMatrix</i>	Generator matrix, saved in 2-dimensional array
<i>N</i>	Length of the code
<i>K</i>	Dimension of the code
<i>Q</i>	Number of elements in the finite field
<i>multiplicativeForm</i>	value=true if the elements of the field are written in multiplicative form, value=false otherwise

Note

The weight distribution is written in the global variable weights

4.2.2.9 calculateWeightDistribution() [3/3]

```
void calculateWeightDistribution (
    int N,
    int K,
    int Q )
```

Parameters

N	Length of the code
K	Dimension of the code
Q	Number of elements in the finite field

Note

The weight distribution is written in the global variable `weights`

4.2.2.10 `detect()`

```
void detect ( ) [extern]
```

It sets the value of `instructionSet` parameter.

Note

ARM architecture is detected by using predefined macros.

4.2.2.11 `find_word_equal_to_fixed_weight()` [1/3]

```
void find_word_equal_to_fixed_weight (
    char * generatorMatrixFile,
    unsigned long long int w_searched )
```

Parameters

<i>generatorMatrixFile</i>	Name of the file, containing the generator matrices
W	value of the searched weight of a codeword

Note

The result for each linear code from the input is written in an output file "find_equal.txt"

4.2.2.12 `find_word_equal_to_fixed_weight()` [2/3]

```
bool find_word_equal_to_fixed_weight (
    int ** generatorMatrix,
    int N,
    int K,
    int Q,
    int W,
    bool multiplicativeForm )
```

Parameters

<i>generatorMatrix</i>	Generator matrix, saved in 2-dimensional array
N	Length of the code
K	Dimension of the code
Q	Number of elements in the finite field
W	value of the searched weight of a codeword
<i>multiplicativeForm</i>	true if the elements of the field are written in multiplicative form

4.2.2.13 find_word_equal_to_fixed_weight() [3/3]

```
bool find_word_equal_to_fixed_weight (
    int N,
    int K,
    int Q,
    int W )
```

Parameters

<i>N</i>	Length of the code
<i>K</i>	Dimension of the code
<i>Q</i>	Number of elements in the finite field
<i>W</i>	value of the searched weight of a codeword

4.2.2.14 find_word_less_than_fixed_weight() [1/3]

```
void find_word_less_than_fixed_weight (
    char * generatorMatrixFile,
    unsigned long long int W )
```

Parameters

<i>generatorMatrixFile</i>	Name of the file, containing the generator matrices
<i>W</i>	value of the searched weight of a codeword

Note

The result for each linear code from the input is written in an output file "find_less_than.txt"

4.2.2.15 find_word_less_than_fixed_weight() [2/3]

```
bool find_word_less_than_fixed_weight (
    int ** generatorMatrix,
    int N,
    int K,
    int Q,
    int W,
    bool multiplicativeForm )
```

Parameters

<i>generatorMatrix</i>	Generator matrix, saved in 2-dimensional array
<i>N</i>	Length of the code
<i>K</i>	Dimension of the code
<i>Q</i>	Number of elements in the finite field
<i>W</i>	value of the searched weight of a codeword
<i>multiplicativeForm</i>	true if the elements of the field are written in multiplicative form

4.2.2.16 find_word_less_than_fixed_weight() [3/3]

```
bool find_word_less_than_fixed_weight (
    int N,
    int K,
    int Q,
    int W )
```


Parameters

N	Length of the code
K	Dimension of the code
Q	Number of elements in the finite field
W	value of the searched weight of a codeword

4.2.2.17 min_dis() [1/3]

```
void min_dis (
    char * generatorMatrixFile )
```

Parameters

<i>generatorMatrixFile</i>	Name of the file, containing the generator matrices
----------------------------	---

Note

minimum distance of the linear codes are written in a output file "min_distance.txt"

4.2.2.18 min_dis() [2/3]

```
unsigned long long int min_dis (
    int ** generatorMatrix,
    int N,
    int K,
    int Q,
    bool multiplicativeForm )
```

Parameters

<i>generatorMatrix</i>	Generator matrix, saved in 2-dimensional array
N	Length of the code
K	Dimension of the code
Q	Number of elements in the finite field
<i>multiplicativeForm</i>	value=true if the elements of the field are written in multiplicative form, value=false otherwise

Returns

minimum distance of the code as unsigned long long int value

4.2.2.19 min_dis() [3/3]

```
unsigned long long int min_dis (
    int N,
    int K,
    int Q )
```

Parameters

N	Length of the code
K	Dimension of the code
Q	Number of elements in the finite field

Returns

minimum distance of the code as unsigned long long int value

4.2.3 Variable Documentation**4.2.3.1 instructionSet**

```
int instructionSet [extern]
```

Note

if value ≥ 9 then the library uses AVX512 instruction set
 if value ≥ 8 then the library uses AVX2 instruction set
 if value ≥ 5 then the library uses SSE4.1 instruction set
 if value < 5 then computations are executed without extended registers

4.2.3.2 weights

```
unsigned long long int weights[] [extern]
```

Global array, containing the (partial) weight distribution of a code, after calculations.

4.3 Data.h File Reference

Declaration of global variables and structures used in the calculations.

Typedefs

- typedef struct _dmatlu_type **dynamic_mat_short**
Definition of dynamic structure describing generator matrix of a linear code saved as dynamic array of unsigned long long int elements.
- typedef struct _dmat_type **dmat_type**
Definition of dynamic structure describing generator matrix of a linear code saved as dynamic array of char elements.

Variables

- **dmat_type bitsCharCF**
Dynamic matrix for calculation when using byte representation.
- **dynamic_mat_short bits**
Dynamic matrix for calculation when using bit representation.
- **dynamic_mat_short matrix**
Dynamic matrix for used to save the read matrix.
- **dynamic_mat_short matrixH**
Dynamic matrix for used for calculation without registers.
- **int test**
Global variable that is used to force the use of a specific instruction set for testing purposes.

4.3.1 Variable Documentation

4.3.1.1 test

```
int test [extern]
```

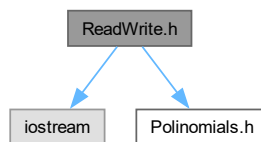
Note

if value > 0, then the value of instructionSet parameter, defined in **LinCodeWeightInv.h** (p. 25), is set to the same value

4.4 ReadWrite.h File Reference

Function declarations for reading and writing code parameters form file and generating random generator matrices in standard form for given parameters of a code.

```
#include <iostream>
#include "Polinomials.h"
Include dependency graph for ReadWrite.h:
```



Functions

- bool **readMatrix** (FILE *fileName, int &n, int &k, int &q)
Function for reading generator matrices from an open file.
- void **printWeights** (unsigned long long int *spec, int N, char *file)
Function for writing the weight spectrum of a linear code in to a file.
- void **printMatrix** (bool form, char *file)
Function for writing the generator matrix in to an output file.
- void **randomgenf** (int n, int k, int q, int num)
Generating random generator matrices of a codes with given parameters.
- void **randomgenf** (int n, int k, int q, int **generatorMatrix, bool multiplicativeForm)
Generating a random generator matrix of a code for given parameters.
- void **write_multpl** (int dec, FILE *write)
Function for writing an element of the field in multiplicative form.
- void **write_multpl_magma** (int dec, FILE *write)
function for writing an element of the field in multiplicative form

4.4.1 Function Documentation

4.4.1.1 printMatrix()

```
void printMatrix (
    bool form,
    char * file )
```

Parameters

<i>form</i>	true, if the elements of the filed are written in the generator matrix in multiplicative form
<i>file</i>	Name of the output file

Note

The generator matrix is read from the global variable **matrix** (p. 35)

4.4.1.2 printWeights()

```
void printWeights (
    unsigned long long int * spec,
    int N,
    char * file )
```

Parameters

<i>spec</i>	Weight spectrum of a code
<i>N</i>	Length of the code
<i>file</i>	Name of the output file

Note

The generator matrix of the code should be written with function printMatrix **printMatrix(bool form, char* file)** (p. 36)

4.4.1.3 randomgenf() [1/2]

```
void randomgenf (
    int n,
    int k,
    int q,
    int ** generatorMatrix,
    bool multiplicativeForm )
```

Parameters

<i>n</i>	Length of the code
<i>k</i>	Dimension of the code
<i>q</i>	Number of elements in the finite field
<i>generatorMatrix</i>	A pointer to a dynamic 2D array that will contain the generator matrices
<i>multiplicativeForm</i>	true if the generator matrix is to be saved in multiplicative form

Note

The generator matrix are written in output file **EXAM**

4.4.1.4 randomgenf() [2/2]

```
void randomgenf (
    int n,
    int k,
    int q,
    int num )
```

Parameters

<i>n</i>	Length of the code
<i>k</i>	Dimension of the code
<i>q</i>	Number of elements in the finite field
<i>num</i>	Number of matrices to be generated

Note

The generator matrices are written in output file **EXAM**. If just one matrix is to be generated, it is saved in global variable **matrix** (p. 35)

4.4.1.5 readMatrix()

```
bool readMatrix (
    FILE * fileName,
    int & n,
    int & k,
    int & q )
```

Parameters

<i>fileName</i>	Pointer to an open input stream of the input file, containing generator matrix
<i>n</i>	Length of the code
<i>k</i>	Dimension of the code
<i>q</i>	Number of elements in the finite field

Returns

true, if error occurs in the input (e.g. there is an character in the generator matrix instead of a number)

Note

if the file is not open, the program end execution with an error code

4.4.1.6 write_multpl()

```
void write_multpl (
    int dec,
    FILE * write )
```

Parameters

<i>dec</i>	Decimal representation of the element
<i>write</i>	Pointer to an open stream for writing

Note

If an incorrect decimal is given, the function will end the program with an error code and an error message will be written in file **error.txt**

4.4.1.7 write_multpl_magma()

```
void write_multpl_magma (
    int dec,
    FILE * write )
```

Function for writing an element of the field in multiplicative form in magma style

Parameters

<i>dec</i>	Decimal representation of the element
<i>write</i>	Pointer to an open stream for writing

Note

If an incorrect decimal is given, the function will end the program with an error code and an error message will be written in file **error.txt**

4.5 Polynomials.h File Reference

Declaration of structure and functions used in computations in composite fields.

```
#include "DataManagement.h"
```

Data Structures

- struct **polinom**
Polynomial structure used for composite finite fields.
- struct **comp_elements**
Structure representing elements of composite finite field.

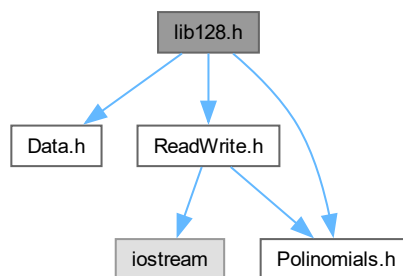
Functions

- void **maketable** (int q)
Functions for generating multiplication tables for a given finite field with q elements

4.6 lib128.h File Reference

Functions for calculations using 128-bit registers.

```
#include "Data.h"
#include "ReadWrite.h"
#include "Polynomials.h"
Include dependency graph for lib128.h:
```



Functions

- void **popcnt_detect** ()
*Function that detects whether the current architecture has a **popcnt** instruction.*
- long long **popcount** (unsigned long long word)
Function that based on the used compiler computes the number of nonzero bits in a computer word.

WeightSpectrum

Calculation of the weight spectrum of a linear code using 128-bit registers

Parameters

generatorMatrix_bits	Generator matrix with appropriate representation of the elements of the field, saved in a dynamic structure, defined in Data.h (p. 34)
n	length of the code
k	dimension of the code
p	characteristic of the field; not present for the functions for field with set characteristic
m	calculations for finite field with p^m elements

Note

The weight spectrum is saved in a global array **weights**

- void **calculateWeightCH3_128** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m)
Function for fields with characteristic 3.
- void **calculateWeightBytes_128** (**dmat_type** &generatorMatrix_byte, int n, int k, int m, int p)
Function for prime and composite fields using byte representation.
- void **calculateWeightCH2_128** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m)
Function for fields with characteristic 2 using bitwise representation.
- void **calculateWeightGF2_128** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k)
Function for GF2 using bitwise representation.

SearchLessThan

Searching for codeword with weight less than given value using 128-bit registers

Parameters

generatorMatrix_bits	Generator matrix with appropriate representation of the elements of the field, saved in a dynamic structure, defined in Data.h (p. 34)
n	length of the code
k	dimension of the code
p	characteristic of the field; not present for the functions for field with set characteristic
m	calculations for finite field with p^m elements
w	the value of the searched weight

Return values

true	if a codeword with weight w is found
------	---

Note

Partial weight spectrum is saved in a global array **weights**

- bool **calculateWeightCH3_128_less_than** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m, int w)
Function for fields with characteristic 3.
- bool **calculateWeightBytes_128_less_than** (**dmat_type** &generatorMatrix_byte, int n, int k, int m, int p, int w)
Function for prime and composite fields using byte representation.
- bool **calculateWeightCH2_128_less_than** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m, int w)
Function for fields with characteristic 2 using bitwise representation.
- bool **calculateWeightGF2_128_less_than** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int w)
Function for GF2 using bitwise representation.

SearchEqualTo

Searching for codeword with weight equal to given value using 128-bit registers

Parameters

generatorMatrix_bits	Generator matrix with appropriate representation of the elements of the field, saved in a dynamic structure, defined in Data.h (p. 34)
n	length of the code
k	dimension of the code
p	characteristic of the field; not present for the functions for field with set characteristic
m	calculations for finite field with p^m elements
w	the value of the searched weight

Return values

true	if a codeword with weight w is found
------	---

Note

Partial weight spectrum is saved in a global array **weights**

- bool **calculateWeightCH3_128_equal** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m, int d)
Function for fields with characteristic 3.
- bool **calculateWeightBytes_128_equal** (**dmat_type** &generatorMatrix_byte, int n, int k, int m, int p, int d)
Function for prime and composite fields using byte representation.
- bool **calculateWeightCH2_128_equal** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m, int d)
Function for fields with characteristic 2 using bitwise representation.
- bool **calculateWeightGF2_128_equal** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int d)
Function for GF2 using bitwise representation.

CountEqualTo

Counting the number of codewords with weight equal to given value using 128-bit registers and writing the generated nonproportional codewords in file **Result_codewords.txt**

Parameters

generatorMatrix_bits	Generator matrix with appropriate representation of the elements of the field, saved in a dynamic structure, defined in Data.h (p. 34)
n	length of the code
k	dimension of the code
p	characteristic of the field; not present for the functions for field with set characteristic
m	calculations for finite field with p^m elements
w	the value of the searched weight
multiplicativeForm	if true, the codewords will be written, using multiplicative representation of the elements of the field

Return values

Total	number of codewords with weight w
-------	--

Note

Weight spectrum is saved in a global array **weights**

- unsigned long long int **calculateNumberOfWordsCH3_128_equal** (**dynamic_mat_short** &generator↔ Matrix_bits, int n, int k, int m, int w, bool multiplicativeForm)
Function for fields with characteristic 3.
- unsigned long long int **calculateNumberOfWordsBytes_128_equal** (**dmat_type** &generatorMatrix_byte, int n, int k, int m, int p, int w, bool multiplicativeForm)
Function for prime and composite fields using byte representation.
- unsigned long long int **calculateNumberOfWordsCH2_128_equal** (**dynamic_mat_short** &generator↔ Matrix_bits, int n, int k, int m, int w, bool multiplicativeForm)
Function for fields with characteristic 2 using bitwise representation.
- unsigned long long int **calculateNumberOfWordsGF2_128_equal** (**dynamic_mat_short** &generator↔ Matrix_bits, int n, int k, int w, bool multiplicativeForm)
Function for GF2 using bitwise representation.

CountLessThan

Counting the number of codewords with weight less than given value using 128-bit registers and writing the generated nonproportional codewords in file **Result_codewords.txt**

Parameters

generatorMatrix_bits	Generator matrix with appropriate representation of the elements of the field, saved in a dynamic structure, defined in Data.h (p. 34)
n	length of the code
k	dimension of the code
p	characteristic of the field; not present for the functions for field with set characteristic
m	calculations for finite field with p^m elements
w	the value of the searched weight
multiplicativeForm	if true, the codewords will be written, using multiplicative representation of the elements of the field

Return values

Total	number of codewords with weight less than w
-------	--

Note

Weight spectrum is saved in a global array **weights**

- unsigned long long int **calculateNumberOfWordsCH3_128_less_than** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m, int w, bool multiplicativeForm)
Function for fields with characteristic 3.
- unsigned long long int **calculateNumberOfWordsBytes_128_less_than** (**dmat_type** &generator↔ Matrix_byte, int n, int k, int m, int p, int w, bool multiplicativeForm)
Function for prime and composite fields using byte representation.
- unsigned long long int **calculateNumberOfWordsCH2_128_less_than** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m, int w, bool multiplicativeForm)
Function for fields with characteristic 2 using bitwise representation.
- unsigned long long int **calculateNumberOfWordsGF2_128_less_than** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int w, bool multiplicativeForm)
Function for GF2 using bitwise representation.

Variables

- unsigned long long int **weights** []
Global variable for that contains weight spectrum.
- int **POPCNT**
*Global variable set by **popcnt_detect()** (p. 43) to show whether the current architecture has a **popcnt** instruction.*
- static const int **N_FIX** = 3072
Constant global variable used for declaration of static arrays, containing different representation of the generator matrix.
- static const int **K_FIX** = 36
Constant global variable used for declaration of static arrays, containing different representation of the generator matrix.
- static short int **TransitionSequence64** [64]
Transition sequences of Q-ary Grey code for field with characteristics 2.
- static short int **TransitionSequence27** [27] = { 0,1,1,2,1,1,2,1,1,3,1,1,2,1,1,2,1,1,3,1,1,2,1,1,2,1,1 }
Transition sequences of Q-ary Grey code for field with characteristics 3.
- static short int **TransitionSequence25** [25] = { 0,1,1,1,1, 2,1,1,1,1, 2,1,1,1,1, 2,1,1,1,1, 2,1,1,1,1 }
Transition sequences of Q-ary Grey code for field with characteristics 5.
- static short int **TransitionSequence49** [49] = { 0,1,1,1,1,1,1, 2,1,1,1,1,1,1, 2,1,1,1,1,1,1, 2,1,1,1,1,1,1, 2,1,1,1,1,1,1, 2,1,1,1,1,1,1, 2,1,1,1,1,1,1 }
Transition sequences of Q-ary Grey code for field with characteristics 7.

4.6.1 Detailed Description

The defined global variables and functions are also used in other places in the library as external components.

4.6.2 Function Documentation

4.6.2.1 popcnt_detect()

```
void popcnt_detect ( ) [extern]
```

Note

This function is used in computations with 128, 256 and 512-bit registers. The implementation is in the corresponding .cpp file.

4.6.2.2 popcount()

```
long long popcount (
    unsigned long long word ) [extern]
```

If POPCNT<0 it uses masks for the calculations.

Parameters

<i>word</i>	A 64-bit computer word of type unsigned long long
-------------	---

Returns

The number of nonzero bits in **word**

Note

This function is used in computations with 128, 256 and 512-bit registers. The implementation is in the corresponding .cpp file.

Table 4.40 Maximum n and k for different fields for $N_FIX = 3072$, $K_FIX = 36$

Field	Formula for N	N	Formula for K	K
$GF(2)$	$N = N_FIX * 8$	24576	$K = K_FIX$	36
$GF(4)$	$N = N_FIX * 4$	12288	$K = K_FIX / 2$	18
$GF(8)$	$N = N_FIX$	3072	$K = K_FIX / 3$	12
$GF(16)$	$N = N_FIX$	3072	$K = K_FIX / 4$	9
$GF(32)$	$N = N_FIX$	3072	$K = K_FIX / 5$	7
$GF(64)$	$N = N_FIX$	3072	$K = K_FIX / 6$	6
$GF(3)$	$N = N_FIX * 4$	12288	$K = K_FIX$	36
$GF(9)$	$N = N_FIX * 2$	6144	$K = K_FIX / 2$	18
$GF(27)$	$N = (N_FIX / 6) * 8$	4096	$K = K_FIX / 3$	12
$GF(p)$	$N = N_FIX * 2$	6144	$K = K_FIX / 2$	18
$GF(p^m)$	$N = N_FIX$	3072	$K = K_FIX / 4$	9

4.6.3 Variable Documentation

4.6.3.1 K_FIX

```
const int K_FIX = 36 [static]
```

This parameter refers to the dimension of the code. The maximum possible dimension of the code depends on this value and the number of elements in the finite field.

Note

value should be multiple of 12

4.6.3.2 N_FIX

```
const int N_FIX = 3072 [static]
```

This parameter refers to the length of the code. The maximum possible length of the code depends on this value and the number of elements in the finite field.

Note

value should be multiple of 1536

Remark: In order to avoid runtime errors we define maximum values of n and k . They depend on the global variable N_FIX and K_FIX and can be easily changed if needed. Table 4.40 shows the maximum values for n and k and the formula used to calculate them for different finite fields.

4.6.3.3 POPCNT

```
int POPCNT [extern]
```

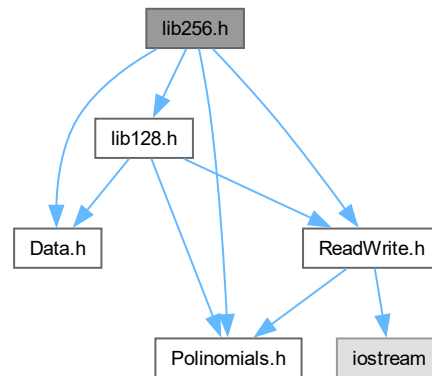
Note

If value < 0 then the architecture does not have a **popcnt** instruction. If value = 1 then the architecture has a **popcnt** instruction for a computer word. If value = 2 then the architecture has a 512-bit **popcnt** instructions. This parameter is used in computations with 128, 256 and 512-bit registers.

4.7 lib256.h File Reference

Functions for calculations using 256-bit registers.

```
#include "Data.h"
#include "lib128.h"
#include "ReadWrite.h"
#include "Polynomials.h"
Include dependency graph for lib256.h:
```



Functions

WeightSpectrum

Calculation of the weight spectrum of a linear code using 256-bit registers

Parameters

generatorMatrix_bits	Generator matrix with appropriate representation of the elements of the field, saved in a dynamic structure, defined in Data.h (p. 34)
n	length of the code
k	dimension of the code
p	characteristic of the field; not present for the functions for field with set characteristic
m	calculations for finite field with p^m elements

Note

The weight spectrum is saved in a global array **weights**

- void **calculateWeightCH3_256** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m)
Function for fields with characteristic 3.
- void **calculateWeightBytes_256** (**dmat_type** &generatorMatrix_byte, int n, int k, int m, int q)
Function for prime and composite fields using byte representation.
- void **calculateWeightCH2_256** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m)
Function for fields with characteristic 2 using bitwise representation.
- void **calculateWeightGF2_256** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k)
Function for GF2 using bitwise representation.

SearchLessThan

Searching for codeword with weight less than given value using 256-bit registers

Parameters

generatorMatrix_bits	Generator matrix with appropriate representation of the elements of the field, saved in a dynamic structure, defined in Data.h (p. 34)
n	length of the code
k	dimension of the code
p	characteristic of the field; not present for the functions for field with set characteristic
m	calculations for finite field with p^m elements
w	the value of the searched weight

Return values

true	if a codeword with weight w is found
------	---

Note

Partial weight spectrum is saved in a global array **weights**

- bool **calculateWeightCH3_256_less_than** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m, int w)
Function for fields with characteristic 3.
- bool **calculateWeightBytes_256_less_than** (**dmat_type** &generatorMatrix_byte, int n, int k, int m, int q, int w)
Function for prime and composite fields using byte representation.
- bool **calculateWeightCH2_256_less_than** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m, int w)
Function for fields with characteristic 2 using bitwise representation.
- bool **calculateWeightGF2_256_less_than** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int w)
Function for GF2 using bitwise representation.

SearchEqualTo

Searching for codeword with weight equal to given value using 256-bit registers

Parameters

generatorMatrix_bits	Generator matrix with appropriate representation of the elements of the field, saved in a dynamic structure, defined in Data.h (p. 34)
n	length of the code
k	dimension of the code
p	characteristic of the field; not present for the functions for field with set characteristic
m	calculations for finite field with p^m elements
w	the value of the searched weight

Return values

true	if a codeword with weight w is found
------	---

Note

*Partial weight spectrum is saved in a global array **weights***

- bool **calculateWeightCH3_256_equal** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m, int d)
Function for fields with characteristic 3.
- bool **calculateWeightBytes_256_equal** (**dmat_type** &generatorMatrix_byte, int n, int k, int m, int q, int d)
Function for prime and composite fields using byte representation.
- bool **calculateWeightCH2_256_equal** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m, int d)
Function for fields with characteristic 2 using bitwise representation.
- bool **calculateWeightGF2_256_equal** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int d)
Function for GF2 using bitwise representation.

CountEqualTo

*Counting the number of codewords with weight equal to given value using 256-bit registers and writing the generated nonproportional codewords in file **Result_codewords.txt***

Parameters

generatorMatrix_bits	Generator matrix with appropriate representation of the elements of the field, saved in a dynamic structure, defined in Data.h (p. 34)
n	length of the code
k	dimension of the code
p	characteristic of the field; not present for the functions for field with set characteristic
m	calculations for finite field with p^m elements
w	the value of the searched weight
multiplicativeForm	if true, the codewords will be written, using multiplicative representation of the elements of the field

Return values

Total	number of codewords with weight w
-------	--

Note

*Weight spectrum is saved in a global array **weights***

- unsigned long long int **calculateNumberOfWordsCH3_256_equal** (**dynamic_mat_short** &generator↵
Matrix_bits, int n, int k, int m, int w, bool multiplicativeForm)
Function for fields with characteristic 3.
- unsigned long long int **calculateNumberOfWordsBytes_256_equal** (**dmat_type** &generatorMatrix_byte,
int n, int k, int m, int q, int w, bool multiplicativeForm)
Function for prime and composite fields using byte representation.
- unsigned long long int **calculateNumberOfWordsCH2_256_equal** (**dynamic_mat_short** &generator↵
Matrix_bits, int n, int k, int m, int w, bool multiplicativeForm)
Function for fields with characteristic 2 using bitwise representation.
- unsigned long long int **calculateNumberOfWordsGF2_256_equal** (**dynamic_mat_short** &generator↵
Matrix_bits, int n, int k, int w, bool multiplicativeForm)
Function for GF2 using bitwise representation.

CountLessThan

*Counting the number of codewords with weight less than given value using 256-bit registers and writing the generated nonproportional codewords in file **Result_codewords.txt***

Parameters

generatorMatrix_bits	Generator matrix with appropriate representation of the elements of the field, saved in a dynamic structure, defined in Data.h (p. 34)
n	length of the code
k	dimension of the code
p	characteristic of the field; not present for the functions for field with set characteristic
m	calculations for finite field with p^m elements
w	the value of the searched weight
multiplicativeForm	if true, the codewords will be written, using multiplicative representation of the elements of the field

Return values

Total	number of codewords with weight less than w
-------	--

Note

Weight spectrum is saved in a global array **weights**

- unsigned long long int **calculateNumberOfWordsCH3_256_less_than** (dynamic_mat_short &generatorMatrix_bits, int n, int k, int m, int w, bool multiplicativeForm)
Function for fields with characteristic 3.
- unsigned long long int **calculateNumberOfWordsBytes_256_less_than** (dmat_type &generatorMatrix_byte, int n, int k, int m, int q, int w, bool multiplicativeForm)
Function for prime and composite fields using byte representation.
- unsigned long long int **calculateNumberOfWordsCH2_256_less_than** (dynamic_mat_short &generatorMatrix_bits, int n, int k, int m, int w, bool multiplicativeForm)
Function for fields with characteristic 2 using bitwise representation.
- unsigned long long int **calculateNumberOfWordsGF2_256_less_than** (dynamic_mat_short &generatorMatrix_bits, int n, int k, int w, bool multiplicativeForm)
Function for GF2 using bitwise representation.

4.7.1 Function Documentation**4.7.1.1 calculateWeightBytes_256()**

```
void calculateWeightBytes_256 (
    dmat_type & generatorMatrix_byte,
    int n,
    int k,
    int m,
    int q )
```

4.8 lib512.h File Reference

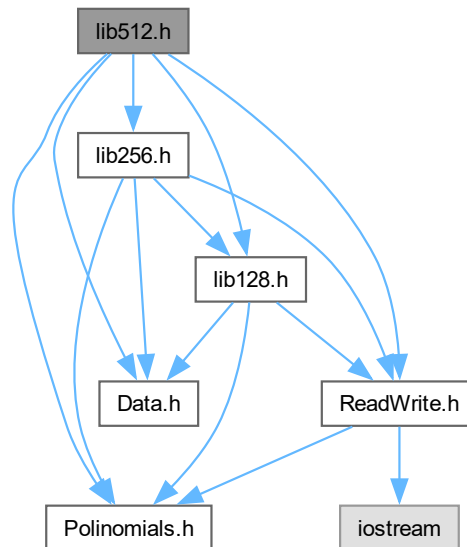
Functions for calculations using 512-bit registers.

```
#include "Data.h"
#include "lib256.h"
#include "lib128.h"
#include "ReadWrite.h"
```



```
#include "Polynomials.h"
```

Include dependency graph for lib512.h:



Functions

WeightSpectrum

Calculation of the weight spectrum of a linear code using 512-bit registers

Parameters

generatorMatrix_bits	Generator matrix with appropriate representation of the elements of the field, saved in a dynamic structure, defined in Data.h (p. 34)
n	length of the code
k	dimension of the code
p	characteristic of the field; not present for the functions for field with set characteristic
m	calculations for finite field with p^m elements

Note

The weight spectrum is saved in a global array **weights**

- void **calculateWeightCH3_512** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m)
Function for fields with characteristic 3.
- void **calculateWeightBytes_512** (**dmat_type** &generatorMatrix_byte, int n, int k, int m, int q)
Function for prime and composite fields using byte representation.
- void **calculateWeightCH2_512** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m)
Function for fields with characteristic 2 using bitwise representation.
- void **calculateWeightGF2_512** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k)
Function for GF2 using bitwise representation.

SearchLessThan

Searching for codeword with weight less than given value using 512-bit registers

Parameters

generatorMatrix_bits	Generator matrix with appropriate representation of the elements of the field, saved in a dynamic structure, defined in Data.h (p. 34)
n	length of the code
k	dimension of the code
p	characteristic of the field; not present for the functions for field with set characteristic
m	calculations for finite field with p^m elements
w	the value of the searched weight

Return values

true	if a codeword with weight w is found
------	---

Note

Partial weight spectrum is saved in a global array **weights**

- bool **calculateWeightCH3_512_less_than** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m, int w)
Function for fields with characteristic 3.
- bool **calculateWeightBytes_512_less_than** (**dmat_type** &generatorMatrix_byte, int n, int k, int m, int q, int w)
Function for prime and composite fields using byte representation.
- bool **calculateWeightCH2_512_less_than** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m, int w)
Function for fields with characteristic 2 using bitwise representation.
- bool **calculateWeightGF2_512_less_than** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int w)
Function for GF2 using bitwise representation.

SearchEqualTo

Searching for codeword with weight equal to given value using 256-bit registers

Parameters

generatorMatrix_bits	Generator matrix with appropriate representation of the elements of the field, saved in a dynamic structure, defined in Data.h (p. 34)
n	length of the code
k	dimension of the code
p	characteristic of the field; not present for the functions for field with set characteristic
m	calculations for finite field with p^m elements
w	the value of the searched weight

Return values

true	if a codeword with weight w is found
------	---

Note

*Partial weight spectrum is saved in a global array **weights***

- bool **calculateWeightCH3_512_equal** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m, int d)
Function for fields with characteristic 3.
- bool **calculateWeightBytes_512_equal** (**dmat_type** &generatorMatrix_byte, int n, int k, int m, int q, int d)
Function for prime and composite fields using byte representation.
- bool **calculateWeightCH2_512_equal** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m, int d)
Function for fields with characteristic 2 using bitwise representation.
- bool **calculateWeightGF2_512_equal** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int d)
Function for GF2 using bitwise representation.

CountEqualTo

*Counting the number of codewords with weight equal to given value using 256-bit registers and writing the generated nonproportional codewords in file **Result_codewords.txt***

Parameters

generatorMatrix_bits	Generator matrix with appropriate representation of the elements of the field, saved in a dynamic structure, defined in Data.h (p. 34)
n	length of the code
k	dimension of the code
p	characteristic of the field; not present for the functions for field with set characteristic
m	calculations for finite field with p^m elements
w	the value of the searched weight
multiplicativeForm	if true, the codewords will be written, using multiplicative representation of the elements of the field

Return values

Total	number of codewords with weight w
-------	--

Note

*Weight spectrum is saved in a global array **weights***

- unsigned long long int **calculateNumberOfWordsCH3_512_equal** (**dynamic_mat_short** &generator↵
Matrix_bits, int n, int k, int m, int w, bool multiplicativeForm)
Function for fields with characteristic 3.
- unsigned long long int **calculateNumberOfWordsBytes_512_equal** (**dmat_type** &generatorMatrix_byte,
int n, int k, int m, int q, int w, bool multiplicativeForm)
Function for prime and composite fields using byte representation.
- unsigned long long int **calculateNumberOfWordsCH2_512_equal** (**dynamic_mat_short** &generator↵
Matrix_bits, int n, int k, int m, int w, bool multiplicativeForm)
Function for fields with characteristic 2 using bitwise representation.
- unsigned long long int **calculateNumberOfWordsGF2_512_equal** (**dynamic_mat_short** &generator↵
Matrix_bits, int n, int k, int w, bool multiplicativeForm)
Function for GF2 using bitwise representation.

CountLessThan

*Counting the number of codewords with weight less than given value using 256-bit registers and writing the generated nonproportional codewords in file **Result_codewords.txt***

Parameters

generatorMatrix_bits	Generator matrix with appropriate representation of the elements of the field, saved in a dynamic structure, defined in Data.h (p. 34)
n	length of the code
k	dimension of the code
p	characteristic of the field; not present for the functions for field with set characteristic
m	calculations for finite field with p^m elements
w	the value of the searched weight
multiplicativeForm	if true, the codewords will be written, using multiplicative representation of the elements of the field

Return values

Total	number of codewords with weight less than w
-------	--

Note

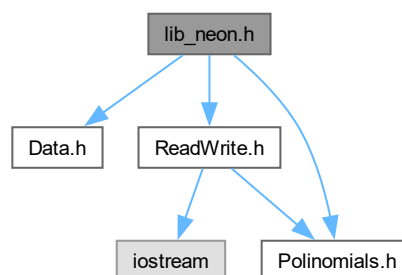
Weight spectrum is saved in a global array **weights**

- unsigned long long int **calculateNumberOfWordsCH3_512_less_than** (dynamic_mat_short &generatorMatrix_bits, int n, int k, int m, int w, bool multiplicativeForm)
Function for fields with characteristic 3.
- unsigned long long int **calculateNumberOfWordsBytes_512_less_than** (dmat_type &generatorMatrix_byte, int n, int k, int m, int q, int w, bool multiplicativeForm)
Function for prime and composite fields using byte representation.
- unsigned long long int **calculateNumberOfWordsCH2_512_less_than** (dynamic_mat_short &generatorMatrix_bits, int n, int k, int m, int w, bool multiplicativeForm)
Function for fields with characteristic 2 using bitwise representation.
- unsigned long long int **calculateNumberOfWordsGF2_512_less_than** (dynamic_mat_short &generatorMatrix_bits, int n, int k, int w, bool multiplicativeForm)
Function for GF2 using bitwise representation.

4.9 lib_neon.h File Reference

Functions for calculations using neon instructions.

```
#include "Data.h"
#include "ReadWrite.h"
#include "Polynomials.h"
Include dependency graph for lib_neon.h:
```



Functions

WeightSpectrum

Calculation of the weight spectrum of a linear code using neon instructions

Parameters

generatorMatrix_bits	Generator matrix with appropriate representation of the elements of the field, saved in a dynamic structure, defined in Data.h (p. 34)
n	length of the code
k	dimension of the code
p	characteristic of the field; not present for the functions for field with set characteristic
m	calculations for finite field with p^m elements

Note

*The weight spectrum is saved in a global array **weights***

- void **calculateWeightCH3_neon** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m)
Function for fields with characteristic 3.
- void **calculateWeightBytes_neon** (**dmat_type** &generatorMatrix_byte, int n, int k, int m, int q)
Function for prime and composite fields using byte representation.
- void **calculateWeightCH2_neon** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m)
Function for fields with characteristic 2 using bitwise representation.
- void **calculateWeightGF2_neon** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k)
Function for GF2 using bitwise representation.

SearchLessThan

Searching for codeword with weight less than given value using neon instructions

Parameters

generatorMatrix_bits	Generator matrix with appropriate representation of the elements of the field, saved in a dynamic structure, defined in Data.h (p. 34)
n	length of the code
k	dimension of the code
p	characteristic of the field; not present for the functions for field with set characteristic
m	calculations for finite field with p^m elements
w	the value of the searched weight

Return values

true	if a codeword with weight w is found
------	---

Note

*Partial weight spectrum is saved in a global array **weights***

- bool **calculateWeightCH3_neon_less_than** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m, int w)
Function for fields with characteristic 3.
- bool **calculateWeightBytes_neon_less_than** (**dmat_type** &generatorMatrix_byte, int n, int k, int m, int q, int w)
Function for prime and composite fields using byte representation.
- bool **calculateWeightCH2_neon_less_than** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m, int w)

Function for fields with characteristic 2 using bitwise representation.

- bool **calculateWeightGF2_neon_less_than** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int w)

Function for GF2 using bitwise representation.

SearchEqualTo

Searching for codeword with weight equal to given value using neon instructions

Parameters

generatorMatrix_bits	Generator matrix with appropriate representation of the elements of the field, saved in a dynamic structure, defined in Data.h (p. 34)
n	length of the code
k	dimension of the code
p	characteristic of the field; not present for the functions for field with set characteristic
m	calculations for finite field with p^m elements
w	the value of the searched weight

Return values

true	if a codeword with weight w is found
------	---

Note

*Partial weight spectrum is saved in a global array **weights***

- bool **calculateWeightCH3_neon_equal** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m, int d)

Function for fields with characteristic 3.

- bool **calculateWeightBytes_neon_equal** (**dmat_type** &generatorMatrix_byte, int n, int k, int m, int q, int d)

Function for prime and composite fields using byte representation.

- bool **calculateWeightCH2_neon_equal** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m, int d)

Function for fields with characteristic 2 using bitwise representation.

- bool **calculateWeightGF2_neon_equal** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int d)

Function for GF2 using bitwise representation.

CountEqualTo

*Counting the number of codewords with weight equal to given value using neon instructions and writing the generated nonproportional codewords in file **Result_codewords.txt***

Parameters

generatorMatrix_bits	Generator matrix with appropriate representation of the elements of the field, saved in a dynamic structure, defined in Data.h (p. 34)
n	length of the code
k	dimension of the code
p	characteristic of the field; not present for the functions for field with set characteristic
m	calculations for finite field with p^m elements
w	the value of the searched weight
multiplicativeForm	if true, the codewords will be written, using multiplicative representation of the elements of the field

Return values

Total	number of codewords with weight w
-------	--

Note

Weight spectrum is saved in a global array **weights**

- unsigned long long int **calculateNumberOfWordsCH3_neon_equal** (**dynamic_mat_short** &generator↔ Matrix_bits, int n, int k, int m, int w, bool multiplicativeForm)
Function for fields with characteristic 3.
- unsigned long long int **calculateNumberOfWordsBytes_neon_equal** (**dmat_type** &generatorMatrix↔ byte, int n, int k, int m, int q, int w, bool multiplicativeForm)
Function for prime and composite fields using byte representation.
- unsigned long long int **calculateNumberOfWordsCH2_neon_equal** (**dynamic_mat_short** &generator↔ Matrix_bits, int n, int k, int m, int w, bool multiplicativeForm)
Function for fields with characteristic 2 using bitwise representation.
- unsigned long long int **calculateNumberOfWordsGF2_neon_equal** (**dynamic_mat_short** &generator↔ Matrix_bits, int n, int k, int w, bool multiplicativeForm)
Function for GF2 using bitwise representation.

CountLessThan

Counting the number of codewords with weight less than given value using neon instructions registers and writing the generated nonproportional codewords in file **Result_codewords.txt**

Parameters

generatorMatrix_bits	Generator matrix with appropriate representation of the elements of the field, saved in a dynamic structure, defined in Data.h (p. 34)
n	length of the code
k	dimension of the code
p	characteristic of the field; not present for the functions for field with set characteristic
m	calculations for finite field with p^m elements
w	the value of the searched weight
multiplicativeForm	if true, the codewords will be written, using multiplicative representation of the elements of the field

Return values

Total	number of codewords with weight less than w
-------	--

Note

Weight spectrum is saved in a global array **weights**

- unsigned long long int **calculateNumberOfWordsCH3_neon_less_than** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m, int w, bool multiplicativeForm)
Function for fields with characteristic 3.
- unsigned long long int **calculateNumberOfWordsBytes_neon_less_than** (**dmat_type** &generator↔ Matrix_byte, int n, int k, int m, int q, int w, bool multiplicativeForm)
Function for prime and composite fields using byte representation.
- unsigned long long int **calculateNumberOfWordsCH2_neon_less_than** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int m, int w, bool multiplicativeForm)
Function for fields with characteristic 2 using bitwise representation.
- unsigned long long int **calculateNumberOfWordsGF2_neon_less_than** (**dynamic_mat_short** &generatorMatrix_bits, int n, int k, int w, bool multiplicativeForm)
Function for GF2 using bitwise representation.

Variables

- static const int **N_FIX** = 3072
Constant global variable used for declaration of static arrays, containing different representation of the generator matrix.
- static const int **K_FIX** = 36
Constant global variable used for declaration of static arrays, containing different representation of the generator matrix.
- unsigned long long int **weights** []
Global variable for that contains weight spectrum.
- static short int **TransitionSequence64** [64]
Transition sequences of Q-ary Grey code for field with characteristics 2.
- static short int **TransitionSequence27** [27] = { 0,1,1,2,1,1,2,1,1,3,1,1,2,1,1,2,1,1,3,1,1,2,1,1,2,1,1 }
Transition sequences of Q-ary Grey code for field with characteristics 3.
- static short int **TransitionSequence25** [25] = { 0,1,1,1,1, 2,1,1,1,1, 2,1,1,1,1, 2,1,1,1,1, 2,1,1,1,1 }
Transition sequences of Q-ary Grey code for field with characteristics 5.
- static short int **TransitionSequence49** [49] = { 0,1,1,1,1,1,1, 2,1,1,1,1,1,1, 2,1,1,1,1,1,1, 2,1,1,1,1,1,1, 2,1,1,1,1,1,1, 2,1,1,1,1,1,1, 2,1,1,1,1,1,1 }
Transition sequences of Q-ary Grey code for field with characteristics 7.

4.9.1 Detailed Description

If ARM architecture is detected by CMake before generation of the IDE project or Makefile, the files **lib128.h** (p. 39), **lib256.h** (p. 45), **lib512.h** (p. 48) and the corresponding .cpp files are not used for the generation of the project. Instead the current file **lib_neon.h** (p. 52) and the corresponding .cpp file are used.

4.9.2 Variable Documentation

4.9.2.1 K_FIX

```
const int K_FIX = 36 [static]
```

This parameter referce to the dimension of the code. The maximum possible dimension of the code depends on this value and the number of elements in the finite field.

Note

value should be multiple of 12

4.9.2.2 N_FIX

```
const int N_FIX = 3072 [static]
```

This parameter refers to the length of the code. The maximum possible length of the code depends on this value and the number of elements in the finite field.

Note

value should be multiple of 1536

Bibliography

- [1] Intel intrinsics guide, 2023. <https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html>.
- [2] ARM. Arm architecture instruction sets guide, 2023. <https://developer.arm.com/architectures/instruction-sets/intrinsics/>.
- [3] LID, R., AND NIEDERREITER, H. *Finite Fields*. Cambridge University Press, 1997.