Available online www.jsaer.com

***Journal of Scientific and Engineering Research*, 2016, 3(1):82-89**

Research Article

ISSN: 2394-2630
CODEN(USA): JSERBR

Comparison of Basic, Bit–at-a-time, and Lookup CRC-32

Mirella A Mioc[1]*, Stefan G Pentiuc[2]

[1]Computer Science Department, "Politehnica" University of Timisoara, Bd. V. Parvan 2, RO-300223, Romania
[2]Faculty of Electrical Engineering and Computer Science, University "Stefan cel Mare", Suceava, Romania

Abstract For keeping the integrity of a message data we can commonly use a Checksum or a CRC (Cyclic Redundancy Code). It is well-known that there is very little information available about their relative performance and effectiveness. Despite that fact we would agree that the choice of the CRC-computing method should be a successful task. For analyze of functioning for common CRCs is possible to use hardware implementation and software implementation. This paper examines the most commonly used CRC software implementations and evaluates their comparative computing time. The software implementations for CRC computing are programs in C language focused on CRC-32 and having as used method: Basic Method, Bit-at-a-time Method and Lookup Method. The contribution of this paper is, concretely, the idea to choose the quickest CRC-computing method.

Keywords Cryptosystem, LFSR (Linear Feedback Shift Register), CRC (Cyclic Redundancy Codes), Irreducible polynomials, Error Detect.

1. Introduction

Cyclic Redundancy Code (CRC) is one of the most widely used error detection technique in data storage devices and serial data communication system to ensure the correctness of the received and stored data. All types of communications use nowadays the CRC calculations. Cyclic redundancy codes handle errors especially for the detection of burst errors. CRCs are widely used in data communications and storage devices. Cyclic redundancy codes use the properties of LFRS functioning in Galois Fields. Data corruption might occur whenever digital data is stored or transmitted. Encoding messages by adding a fixed – length check value, for the purpose of error detection in communication networks are used in systematic cyclic codes. The well-known 32–bit generator polynomial is used in Ethernet and many other standards. A very important goal in this frame is using of cryptosystems [1]. Beginning with the AES (Advanced Encryption Standard) [2], all the Encryption and Decryption Algorithms developed the safe work in networks. CRCs (Cyclic redundancy codes) preserve the integrity of data in transmission and storage applications. CRC can be implemented by using either software or hardware methods. A circuit based on a simple shift register is enough for obtaining the necessary computations in a traditional hardware implementation. The other kind of implementation, meaning the software implementation, handling the data (as bytes or even words) becomes more convenient and faster. In the literature several software algorithms have been reported. Usually each algorithm can be analyzed and compared from the point of view of the speed and storage requirements. In all applications based on working in a Galois Field [3] is necessary to use irreducible polynomials [4] for increasing the security [5]. Some other aspects revealed by using LFSR are in the field of testability [6].

2. Description of the analyze

Cyclic codes are often used in computer networks for their high error correcting capabilities.

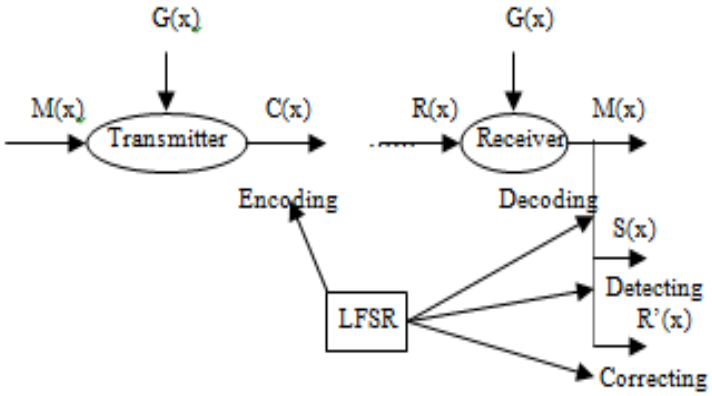

Figure 1: Error control scheme using LFSR as Basic block

G(x) - generator polynomial
M(x) - message to send
R(x) - received message
C(x) - code word (converted M(x))
E(x) - error polynomial
S(x) - syndrome extracted
R`(x) - code word (converted R(x))

$$R(x) = C(x) + E(x) \quad (1)$$

An error is detected if $S(x) \neq 0$ and $E(x) \neq 0$

LFSRs can be used as a random number generator [7]. In this case the sequence is a pseudo - random sequence, characterized through the fact that the numbers appear in a random sequence and repeats every 2^n-1 patterns. Cryptography [8], automatic testing computer graphics and others are based on the random number generator [9].

One of the most important uses is the one from the field of error detection and correction [10].

Table 1: Generator Polynomials of Some Common CRCs

S. No.	Generator Polynomial	Hex	Name	Degree
1.	$x^{12}+x^{11}+x^3+x^2+x+1$	80F	CRC-12	12
2.	$x^{16}+x^{12}+x^5+1$	1021	CRC-CCITT	16
3.	$x^{16}+x^{15}+x^2+1$	8005	CRC-16	16
4.	$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x7+x5+x^4+x^2+x+1$	04C1 1DB7	CRC-32	32

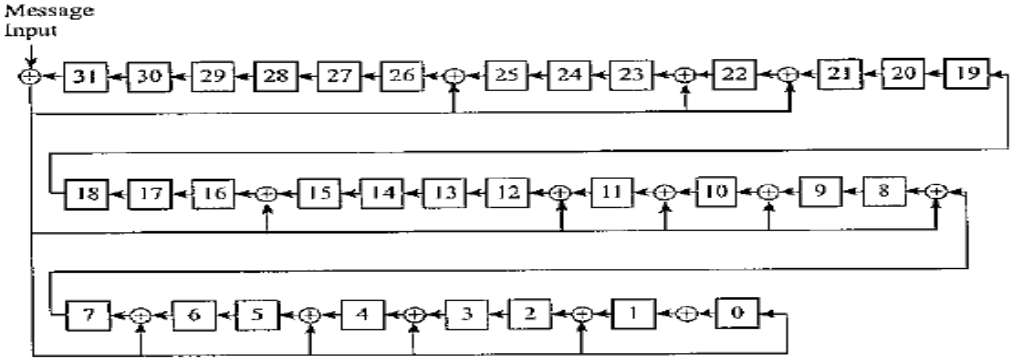

Figure 2: Scheme for CRC – 32

Some other aspects about increasing the security are developed in using Digital Signature [11]. Also a modern and efficient method in security is the use of free distance of convolutional codes [12]. CRC-CCITT is a well-known CRC used by the Comité Consultatif Internationale Téléphonique.

In the terminology used in CRC computation the following terms are presented:
For protecting Data Integrity, it is possible to use CRCs and Checksums. Generally speaking, we have the data to transmit, which is necessary to be protected. Often this data can have dimensions as M bytes and is called Data Word. The resulting sequence obtained by using CRC or Checksum calculation is named Check Sequence or Frame Check Sequence (FCS). The new word containing the Data Word and the Check Sequence taken together became the Code Word.

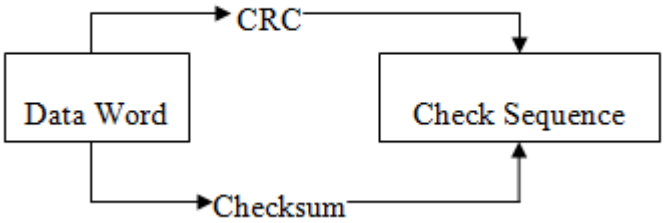

Figure 3: The Code Word

To verify if the data was not corrupted, the result of the calculation must be equal to the Check Sequence. Analyzing these two possibilities to obtain the Check Sequence a problem appears: which is the best. Compute CRC is more difficult than Checksum because the first one detects more possible bit errors than the second one which detects only 1 and 2-bit errors. Much more than this is to find a good CRC.
Hamming Distance (HD) between two bit strings is the number of bits for changing to convert one to the other.

3. Experimental results

Many case studies focused on analyzing the error detection capabilities obtained by using the common polynomials. For analyzing the functioning of different common CRCs some software implementations were used [13]. All these methods are based on working with bit operations. Each of the used CRCs are handled as hexa symbols, meaning that for the generator polynomial

$$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^{8}+x7+x5+x^{4}+x^{2}+ x+1 \quad (2)$$

the scheme is presented in figure 3 and the coefficients of the polynomial and the following:

100000100110000010001110110110111

meaning 04C11DB7 in hexa (the first 1 is missing).

In the next three figures, the implementations are presented and represent programs in C language for the following methods:

- Basic CRC-32;
- Bit-at-a-time CRC-32;
- Lookup CRC.

Here we have three implementations of CRC. Below will be three functions: CRC32_v1, CRC32_v2 and CRC32_v3.

We start with the first function, CRC32_v1.

```
/* CRC-32C (iSCSI) polynomial
in reversed bit order. */
#define POLY 0x82f63b78

uint32_t crc32_v1(uint32_t crc,
    const char *buf,
    size_t len)
{
    int k;

    crc = ~crc;
    while (len--) {
        crc ^= *buf++;
        for (k = 0; k < 8; k++)
            crc = crc & 1 ? (crc >> 1)
            ^ POLY : crc >> 1;
    }
    return ~crc;
}
```

This function was tested with Intel® VTune™ Amplifier to show the basic hotspots.
Below is a screenshot with CPU Time of all functions which are in the CPP file including our function, CRC32_v1.

CPU Time: Total	CPU Time: Self	Function (Full)
1.024s	0s	
1.024s	0s	func@0x4b2e5673
1.024s	0s	func@0x4b2e5694
1.024s	0s	BaseThreadInitThunk
1.024s	0s	mainCRTStartup
1.024s	0s	_scrt_common_main
1.024s	0s	_scrt_common_main_seh
1.023s	0s	invoke_main
1.023s	0.003s	main
0.564s	0.564s	std::basic_ostream<char,struct ...
0.452s	0.452s	crc32_v1(unsigned int,char con ...

Figure 4: CRC32_v1 function execution time

Our function takes 0.425 seconds to execute.
When the testing module is running the function is called as many times as possible in a five seconds interval. Basically the call of that function is located in a *while block*, an example is showed below.

```
int main() {
    clock_t start;
    double duration;

    start = clock();
    clock_t now = start;
    const int DURATION_IN_MILLIS = 5000;

    while ((now - start) < DURATION_IN_MILLIS) {
     char chars[] = "Lorem ipsum dolor sit amet, cc

        uint32_t result = crc32_v3(chars);
        cout << result;

        now = clock();
    }
}
```

Next we have this function, CRC32_v2.

```
unsigned int crc32_v2(char *mess) {
    int  i, j;
    unsigned int oct, crc, masca;
    static unsigned int tab[256];

        if(tab[1] == 0) {
            for (oct = 0; oct <= 255; oct++)  {
            crc = oct;
            for (j = 7; j >= 0; j--) {
                masca = ~(crc & 1);
                crc = (crc >> 1) ^ (0xEDB88320 & masca);
            }
            tab[oct] = crc;
    }
        }
    i = 0;
    crc = 0xFFFFFFFF;
        while((oct = mess[i]) != 0) {
            crc = (crc >> 8) ^ tab[(crc^ oct) & 0xFF];
            i = i + 1;
        }
    return  ~crc;
}
```

And the results for this function are:

CPU Time: Total	CPU Time: Self	Function (Full)
815.117ms	0ms	
815.117ms	0ms	func@0x4b2e5673
815.117ms	0ms	func@0x4b2e5694
815.117ms	0ms	BaseThreadInitThunk
815.117ms	0ms	mainCRTStartup
815.117ms	0ms	_scrt_common_main
815.117ms	0ms	_scrt_common_main_seh
814.668ms	0ms	invoke_main
814.668ms	4.910ms	main
750.959ms	750.959ms	std::basic_ostream<char,stru ...
52.145ms	52.145ms	crc32_v2(char *)

Figure 5: CRC32_v2 function execution time

This function takes 0.052145 seconds to execute.
Finally this is the CRC32_v3 implementation:

```
unsigned int crc32_v3(char *mess) {
    int  i, j;
    unsigned int oct, crc, masca;

    i = 0;
    crc = 0xFFFFFFFF;
    while (mess[i]  != 0) {
        oct = mess[i];
        crc = crc ^ oct;
        for (j = 7; j >= 0; j--) {
            masca = ~(crc & 1);
            crc = (crc >> 1) ^ (0xEDB88320 & masca);
        }
        i = i + 1;
    }
    return ~crc;
}
```

Testing result for this function:

CPU Time: Total	CPU Time: Self	Function (Full)
969.447ms	0ms	
969.447ms	0ms	func@0x4b2e5673
969.447ms	0ms	func@0x4b2e5694
969.447ms	0ms	BaseThreadInitThunk
969.447ms	0ms	mainCRTStartup
969.447ms	0ms	_scrt_common_main
969.447ms	0ms	_scrt_common_main_seh
969.447ms	0ms	invoke_main
969.447ms	5.507ms	main
587.901ms	587.901ms	std::basic_ostream<char,str ...
360.033ms	360.033ms	crc32_v3(char *)

Figure 6: CRC32_v3 function execution time

This function takes 0.360033 seconds to execute.

For the purpose of accuracy these are the CPU Specs.

Table 2: CPU Specifications

Name	**3rd generation Intel(R) Core(TM) Processor family**
Frequency	2.2 GHz
Logical CPU Count	8

Bellow there is a diagram that shows the average use of CPU and Memory.

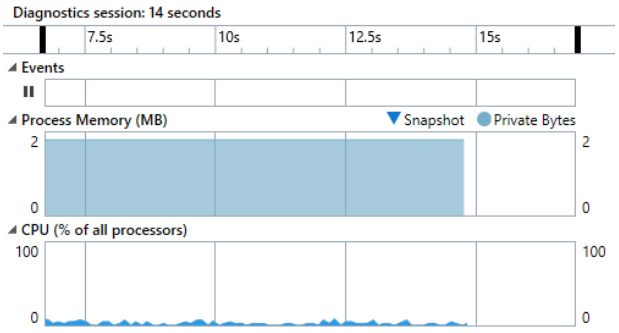

Figure 7: Average use of CPU and Memory

Clearly we can see that average use of memory is approximately 2 MB and CPU use level is ~3.5%.

Now, let's break down the data. Here we have a graph with execution time of all three functions.

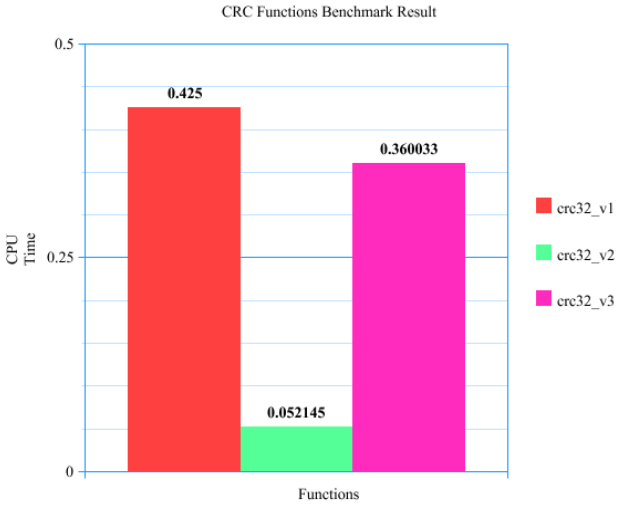

Figure 8: Execution time of the three functions

As can be seen from the figure above the computing time for the three computing methods are quite different: the best one is for the Bit-at-a-time Method. This graph tells us that the less time consuming implementation is CRC-32_v2 with 0.052145 seconds. One of the most important aspect of the analyze is obtaining the Hamming Distance. Usually a given Hamming Distance is proposed for being achieved. In the table 3 some HD for common CRCs are presented. Other important used values are the values for the initial remainder and the Final XOR obtained value, which is shown in the table 4.

Table 3: Hamming Distance for some common CRCs

S. No.	Polynomial	Hex	HD
1.	$x^{32}+x^8+x^7+x^6+x^4+x^2+x^1+x^0$	0x000001D7	8
2.	$x^{32}+x^8+x^6+x^5+x^4+x^3+x^0$	0x00000179	7
3.	$x^{32}+x^8+x^7+x^6+x^5+x^3+x^2+x^0$	0x000001ED	6
4.	$x^{32}+x^7+x^6+x^5+x^2+x^0$	0x000000E5	6

Table 4: Basic Information for the common CRCs

	Polynomial	Width	Initial Remainder	Final XOR Value
CRC-CCITT	0x1021	16 bits	0xFFFF	0x0000
CRC-16	0x8005	16 bits	0x0000	0x0000
CRC-32	0x04C11DB7	32 bits	0xFFFFFFFF	0xFFFFFFFF

There are many researches focused on choosing the best used CRC polynomials for providing a capability to obtain a less error detection. Beyond this, some other researches had the goal to find the best polynomials providing good performance for different data word lengths.

For such analyze is necessary to take into account the weight of the polynomials, the data word length (in bits), effectively the polynomial, and the first weights.

The HD=1 weight is always zero and because of that it is omitted, so that the first weight given is for HD =2 (2-bit errors in the Code Word).

The analyze of the polynomial

$$x^8+x^5+x^2+x^1+x^0 \qquad (3)$$

having for data words of 18 to 55 bits the HD=4, will obtain all 1-, 2-, and 3-bit errors for these lengths.

4. Conclusions

In the fight against data corruption, one of the first places is occupied by the CRC. A CRC has a special place in detecting data corruption. From mathematical point of view, a CRC uses polynomial division and arithmetic over the field of integers mod 2.

This paper explored different implementation choices for computing one of the most widely used CRC codes.

A complete analyze of all billion possible 32 - bit polynomials is a hard task even for their enumeration and using all existing filtering and computational techniques.

Almost all of the commonly used CRC polynomials provide significantly less error detection capability than they might.

Also, the analyze reveals that this polynomials are only good choices for particular message lengths.

For a CRC, the HD depends on the generator polynomials used, on the data word length, and the FCS length.

The comparing software variants approaches can be used with different kind of processors.

Our proposed point of view is to choose the quickest CRC computing method.

References

[1]. Niederreiter, H. (1985, April). A public-key cryptosystem based on shift register sequences. In *Advances in Cryptology—EUROCRYPT'85* (pp. 35-39). Springer Berlin Heidelberg.

[2]. Daemen, J., & Rijmen, V. (2013). *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media.

[3]. Scott, M. (2007). Optimal Irreducible Polynomials for GF (2m) Arithmetic. *IACR Cryptology ePrint Archive*, *2007*, 192.
[4]. Udar, S., & Kagaris, D. (2007, July). LFSR reseeding with irreducible polynomials. In *On-Line Testing Symposium, 2007. IOLTS 07. 13th IEEE International* (pp. 293-298). IEEE.
[5]. Mioc, M. A. (2008, September). Study of using shift registers in cryptosystems for grade 8 irreducible polynomials. In *WSEAS Conference SMO* (pp. 23-25).
[6]. Al–Yamani II, A. (2002). Logic BIST: Theory, Problems, and Solutions. Stanford University, RATS/SUM02.
[7]. Aguirre, J. V., Álvarez, R., Tortosa, L., & Zamora, A. (2008). An optimized pseudorandom generator using packed matrices. *WSEAS Transactions on Information Science and Applications*, *5*(4), 487-496.
[8]. Schneier, B. (1996). Applied cryptography: protocols. *Algorithms, and Source Code in C*, *2*, 216-222.
[9]. Sedaghat, R., & O'Brien, B. (2003). ASIC Implementation of a Pseudo-random Test Pattern Generator Using a 32-bit Linear Feedback Shift Register (LFSR). In *Proc. International Conference for Upcoming Engineers, ICUE*.
[10]. Peterson, W. W., & Weldon, E. J. (1972). *Error-correcting codes*. MIT press.
[11]. Alvarez, R., Martinez, F. M., Vicent, J. F., & Zamora, A. (2008). A matricial public key cryptosystem with digital signature. *A A*, *1*, 1.
[12]. Bose, R. (2006). An efficient method to calculate the free distance of convolutional codes. *WSEAS Transactions on Electronics*, *3*(10), 525.
[13]. Mioc, M. A. (2012) Analyze of common CRCs functioning using software implementation. *Advances in Data Networks, Communications, Computers and Materials*, 192-196.