A review of CRC mechanisms and FOSS algorithms

Mahdi Amiri-Kordestani[1], Afaq Ahmed[2], Hadj Bourdoucen[3]

Abstract— The aim of this paper is to provide an overview for cyclic redundancy check method known as CRC mechanism applied to some popular FOSS applications. This is done through reviewing a number of notable papers published during the last fifty years. Major implementations and open source algorithms including bit-wise, byte-wise, lookup table, slicing-by-4 and 8 have been studied and presented.

Index Terms— CRC, Cyclic Redundancy Check, Fault tolerant, FOSS Algorithms

I. INTRODUCTION

Error checking codes are mechanisms to make sure data has been transmitted or stored correctly. In other words, they are used for investigating data integrity [1]. There are many different integrity mechanisms, from simple parity checks to advanced hash functions but Cyclic Redundancy Check (CRC) is a simple, fast, yet powerful common mechanism introduced by Peterson[2] in 1961 which is based on the remainder of a polynomial division in the finite field theory based on cyclic codes. Peterson introduced two Linear Feedback Shift Register circuit to implement CRC and his method is known as bit-wise CRC. Around 1983, byte-wise CRC algorithm was introduced by Perez[3] for faster calculation of CRC mechanism in computers by software. Latter, CRC Look-up tables were introduced by Ramabadran[4] and Sarwate[5] around 1988 which are still in use by embedded and general computing systems. An attempt towards Parallel CRC was started by Albertengo[6] in 1990 to produce a method to calculate CRC in parallel. Sometime later, Williams, Ross N. developed a generic and portable fast C algorithm on 1993 based on Look-up tables which is still in use at FreeBSD kernel operating system. After the nineties, there were many researches done on CRC but notable research by Castagnoli[7] and Koopman[8, 9] led the CRC to selecting better polynomials. Around 2010, Intel[10] and AMD introduced Carry-less Multiplication (CLMUL) which improved computation of CRC on generic computers. Nowadays CRC is used in many communication protocols like Ethernet, iSCSI, Bluetooth, ATM, mobile networks, synchronous data link control(SDLC), high level data link control (HDLC), IrDa, USB, MMC storage and etc.

Terminology

The following abbreviations have been used in this paper:

- **G(x)**: Generator (used in CRC dividing) in form of a polynomial.
- **M(x)**: Message (a fixed size of data) in form of a polynomial.
- **R(x)**: CRC check bits in form of a polynomial.
- **T(x)**: CRC encoded message in form of a polynomial.
- **H(x)**: CRC encoded message with errors in form of a polynomial.
- **E(x)**: Error pattern in form of a polynomial.

II. THEORY BEHIND CRC

A. Polynomial representation

The idea behind CRC comes from the well-known cyclic codes where the encoded data should be partitioned to fixed size blocks. For instance, consider each binary block as a message containing k bits of data. Hence, a block of 8 bits will have 8 bits of either 0 or 1. Each bit from the most important bit from left side (could be right) can be a coefficients for a term variable x^k where k is the position of the bit in the block starting from the first bit at the right in position 0 to the last bit at the left in position $k-1$. For instant, a block of 11110001 has 8 bits and represents an eight-term polynomial with coefficients of 1, 1, 1, 1, 0, 0, 0 and 1:

- High order to low order:
  $1x^7 + 1x^6 + 1x^5 + 1x^4 + 0x^3 + 0x^2 + 0x^1 + 1x^0$
- Low order to high order:
  $1x^0 + 1x^1 + 1x^2 + 1x^3 + 0x^4 + 0x^5 + 0x^6 + 1x^7$

The order in representing data as polynomials depends on the hardware, the way data is going to be transmitted, retrieved or stored and is described by term Endianness. In Big-Endian the most important byte (normally the byte from left hand side) is stored at lowest address where in Little-Endian, the least important byte is stored at the lowest address. Bit-Endians or network-order describes the way the binary data is going to be sent, Ethernet and RS-232 send the low bit first (Little-Endian). [1] [12].

B. Mathematics of CRC

To simplify the mathematics and hardware implementations, CRC is using modulo 2 rules of algebraic finite field theory which is also known as Galois field (GF) in all arithmetic operations. In GF(2) theory, there are only two elements: *0* and *1* and to simplify the operations, the carry of adding and subtracting will be omitted, so: $1x^n + 1x^n = 0x^n$, $1x^n - 1x^n = 0x^n$, $1x^n = -1x^n$. Both addition and subtracting operations are equal to logical exclusive-or operation (XOR) which can be

[1] Communication and Information Research Center (CIRC), Sultan Qaboos University, P.O. 17, Al-Khodh, Muscat-123, Sultanate of Oman (phone: 968-24142863; fax: 968-24144336; e-mail: amiri@squ.edu.om)

[2] Department of Electrical and Computer Engineering, College of Engineering, Sultan Qaboos University, P.O. 33, Al-Khodh, Muscat -123, Sultanate of Oman; (phone: 968-24141327; fax: 968-24413454; e-mail: afaq@squ.edu.om

[3] Communication and Information Research Center (CIRC), Sultan Qaboos University, P.O. 17, Al-Khodh, Muscat-123, Sultanate of Oman (phone: 968-24143696; fax: 968-24144336; e-mail: hadj@squ.edu.om)

 ISSN:1813-419X

simply implemented in hardware by using combination of shift registers and XOR gates.

In order to encode data in CRC, both parties (for example the sender and receiver) should first agree on a fixed polynomial called the generator G(x). For example the generator of CRC-4-ITU is the polynomial of $G(x) = x^4 + x + 1$. This polynomial has a degree of 4; where in binary format it is 10011 and has 5 bit length. In CRC terminology, CRC − n referrers to a polynomial of degree n; for instance, CRC − 32 which is used in Ethernet has a polynomial with degree of 32 with a maximum length of 33 bit length for its generator and 32 bit length for appended CRC check bits.

To generate the CRC encoded message, the first step is to portion data to fixed sized blocks. The block size length can be from one byte to several gigabytes. Each block of data is called *M(x)* and represents a polynomial with a degree of *k−1*, where *k* is the length of the block in bits. The rest is to multiply *M(x)* by x^r, where *r* is the degree of polynomial *G(x)* and then divide the result by the generator *G(x)*. The remainder of such division is called *R(x)* and is the CRC check bits of *M(x)* by *G(x)*. The final step is to add the remainder *R(x)* to $x^rM(x)$.

The calculation can be summarized in the following steps:

1. Data will be portioned to blocks of k bit length, each block will generate a message of *M(x)* which is a polynomial of degree $k - 1$.
2. Calculation of $x^rM(x) = G(x)Q(x) + R(x)$ will be done on each block. Where r is the degree of the generator *G(x)*, and *Q(x)* is the quotient which will be omitted and *R(x)* is the remainder.
3. Encoded message will then be $F(x) = x^rM(x) + R(x)$.

As an example, assume a data with length of 1024 bits. If we want to use CRC-4-ITU with the generator of $G(x) = x^4 + x + 1$ which has a degree $r = 4$, we can divide the data to blocks of 8 bits. Suppose the first block contains 11110001 and the polynomial is $M(x) = x^7 + x^6 + x^5 + x^4 + 1$. Then $x^rM(x)$ will become $x^{11} + x^{10} + x^9 + x^8 + x^4$ and the remainder of dividing $x^rM(x)$ by *G(x)* in modulo 2 would be $R(x) = x^2 + 1$. Finally the encoded message will have the following form:

$T(x) = x^rM(x) + R(x) = x^{11} + x^{10} + x^9 + x^8 + x^4 + x^2 + 1$

Figure 1 CRC Encoded Message

Additional explanation on CRC mathematics can be found in [4], [1] and [2].

C. Common Types of Errors

The novel aspect of CRC resides in its ability to detect common transmission errors consisting erroneous data. Assume one has sent the encode message T(x) and received the message with errors, *H(x)* where *T(x) <>H(x)*. We can write *H(x) = T(x) + E(x)* where *E(x)* would be the error pattern. If datahad only one erroneous bit *(E(x) = x^i)*, the error is named "Single Error". But, if there are two erroneous bits *(E(x) = $x^i + x^j$*where*i> j)*, the error is named "Double Error". Similarly we can define "Triple Error" where there are tree erroneous bits ($E(x) = x^i + x^j + x^k$ where $i > j > k$) and so on.

A "burst error" is a series of continues erroneous bits which can be defined as:

$$E(x) = x^m \sum_{0}^{n-1} x^i = x^{m+n-1} + x^{m+n-2} + \ldots + x^m$$

Where *n* is the number of erroneous bits, and *m* is the place of burst error from the right hand side.

Another interesting topic in data integrity is Hamming Distance (HD). In coding theory,HD shows the minimum number of erroneous bits which cannot be detected by a mechanism. For example, HD of one means there can be an undetectable single error. In general, HD of *k* means, there can be *k* erroneous bits in the message which are not detectable by mechanism while all combinations of $k - 1$ and less erroneous bits are detectable.

D. CRC Data Integrity

To check the integrity of data, the receiver divides T(x) by G(x):

$$\frac{T(x)}{G(x)} = \frac{x^rM(x) + R(x)}{G(x)}$$

As $x^rM(x)$ have already defined previously as $M(x) = G(x)Q(x) + R(x)$, so we will have:

$$\frac{T(x)}{G(x)} = \frac{G(x)Q(x) + R(x) + R(x)}{G(x)} = Q(x)$$

It's obvious if the remainder is zero *(R(x) = 0)*, the receiver will not detect any error butby choosing a good generator G(x), CRC mechanism can detect a variety of errors including single errors, double errors and some burst errors as well. We have the following operation in case of errors:

$$\frac{H(x)}{G(x)} = \frac{T(x) + E(x)}{G(x)} = \frac{x^rm(x) + R(x) + E(x)}{G(x)} = Q(x) + \frac{E(x)}{G(x)}$$

The remainder of dividing *H(x)* by *G(x)* will become zero only if the remainder of dividing *E(x)* by *G(x)* is also zero. For all detectable errors, this remainder should not be zero; otherwise the CRC mechanism will not detect it. In other words, for all undetectable errors one should have:

$$E(x) = Q_e(x)G(x).$$

In a single error case, we have $E(x) = x^i$, so we may have $x^i = Q_e(x)G(x)$. If a polynomial *G(x)* contains two terms like $x + 1$, the equality cannot be correct which show that all polynomials *G(x)* with more than one term (like $x + 1$), can identify single errors.

For double error case, we have $E(x) = x^i + x^j$ where $i > j$, so we may have:

$$x^i + x^j = Q_e(x)G(x)$$
$$x^j(x^{i-j} + 1) = Q_e(x)G(x)$$

And for all burst errors case, wewe'll have:

$$E(x) = x^m \sum_{0}^{n-1} x^i = Q_e(x)G(x)$$

As described earlier, by choosing a good polynomial for CRC generator, it can detect all burst errors less than the degree of its generator polynomial and can detect all errors on the odd number bits of the message. Some examples and mathematical proof on how CRC can detect these errors has been firstly introduced in [2] and also explained in a number of papers, e.g. [4] and [1].

III. CRC IMPLEMENTATION

Traditional hardware implementation of CRC was based on a simple shift register circuits that handled the data bit by bit. By using computers instead of computing bit by bit, it's much faster to work with bytes, words or double words. A faster solution is to have or create a look-up table in advance and use that table instead of real computing.

Different standards on CRC vary on size of message, generator polynomial, reversing the message or polynomial bits, an initial value for remainder and also an additional mask for the final remainder. Some of the common CRC versions are CRC-8, CRC-12, CRC-16 and CRC-32. Not all standards or telecommunication systems are using the same generator polynomials, for example CRC-8 has all the following popular polynomials:

- CRC-8 ($x^8 + x^7 + x^6 + x^4 + x^2 + 1$)
- CRC-8-ATM ($x^8 + x^2 + x + 1$)
- CRC-8-CCITT ($x^8 + x^7 + x^3 + x^2 + 1$)
- CRC-8-Dallas/Maxim ($x^8 + x^5 + x^4 + 1$)
- CRC-8-SAE J1850 ($x^8 + x^4 + x^3 + x^2 + 1$)

Probably the most famous CRC is CRC-32 which is used in the Ethernet. Researchers have shown that there are many alternate generator polynomials which could be used in CRC-32 method to improve the error detecting efficiency [8][11].

CRC-32C uses a better polynomial and was proposed to be used in iSCSI in RFC3385 around 2002. Nowadays, both Intel and AMD support SSE4.2 instructions in their modern chips and do the most of CRC-32C mechanism directly in hardware level with just a few lines of assembly code[10].

A. Bit-wise Algorithm

CRC bit-wise (CRCB) algorithm is the simplest way to compute CRC and is based on a simple shift register circuit logic. For example, the following is a hardware implementation of polynomial $G(x) = x^4 + x + 1$ with LFSR (Linear Feedback Shift Register) including D-flip-flops and XOR gates. See Figure 2.

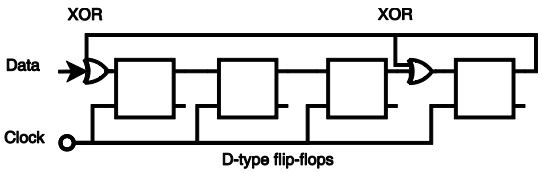

Figure 2: CRC-4-ITU with LFSR

The flowchart of CRCB is also very simple to understand and implement. Based on CRC's polynomial division, XOR operation will be executed between the message and generator starting from the left hand message bit (MSb) to the right hand bit (LSb) only if MSb is not zero (message is divisible). To achieve this, a simple loop will check the MSb first and if it is not zero, it will execute the XOR operation, then the message will be shifted left once, and if message is not finished yet, the loop repeats. See Figure 3.

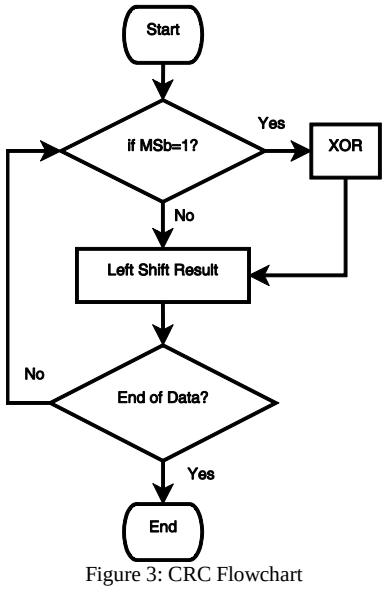

Figure 3: CRC Flowchart

CRCB can be easily implemented in C and other programming languages. The Algorithm 1will encode a single character with polynomial $G(x) = x^4 + x + 1$. The algorithm in using "unsigned char" for storing the polynomial value and can handle generator polynomials with maximum degree of 7. The code in Algorithm 1 is based on the code provided at [12] and is public domain.

```
#define POLYNOMIAL 0x98 /*10011000*/
unsigned char
crc(unsigned char const message)
 {
unsigned char remainder,bit;
remainder = message;
for (bit = 8; bit > 0; --bit)
 {
if (remainder & 0x80)
        remainder ^= POLYNOMIAL;
remainder = (remainder << 1);
 }
return (remainder >> 4);
}
```

Algorithm 1: CRC C source code

B. Lookup Table Algorithm

To archive faster computation of CRC mechanism, we can create a lookup table for all possible values of a message in advance and use the lookup table instead of recalculating CRC every time. For a message *M(x)* containing *k* bits of data, we will need a table with 2^k rows to store all possible values. The Table 1 demonstrates the CRC check value of polynomial $G(x) = x^4 + x + 1$ for a single byte message.

Value	CRC Check
0	0
1	3

2	6
3	5
4	12
5	10
...	...
255	4

Table 1: CRC Lookup check bits (decimal)

We can easily create and manipulate the table by using a single array of size 2^k. The code in Algorithm 2 will create the table for all possible single byte messages (k = 8) by calling the function defined in Algorithm 1.

```
unsigned char crcTable[256];
voidcrcInit()
{
for (inti=0;i<256;i++)
        crcTable[i]=crc(i);
};
```

Algorithm 2: CRC Lookup Table

By using array "crcTable" as a lookup table, to get the CRC remainder of M(x), we can easily read the value crcTable[M(x)]. More explanation and better algorithm can be found on [5] which is very popular in general and embedded systems.

C. More implementations

Although the Algorithm shown in 1 works fine on a single byte message, but one cannot load a very long message directly to registers in order to calculate the XOR value & we need to slice the data to smaller parts. Assume $M_L(x)$ is a long message which we can write it as $M_L(x) = M_1(x):M_2(x)$. Here the ":" operation means we have sliced $M_L(x)$ is two parts $M_1(x)$ and $M_2(x)$. We can say:

$$[M_1:M_2] \bmod G = [x^k M_1(x) + M_2(x)] \bmod G =$$
$$[x^k(Q_1(x)G(x) + R_1(x)) + M_2(x)] \bmod G =$$
$$[x^k Q_1(x)G(x)] \bmod G + [x^k R_1(x) + M_2(x)] \bmod G =$$
$$[R_1:M_2] \bmod G$$

Based on the theorem above, one can divide long data to smaller pieces of data and calculate the CRC of each part individually. This is the main idea behind faster implementations of CRC and parallel CRC approaches. Two modern implementations are the slicing-by-4 and slicing-by-8 algorithms from [13] which have a significant speed improvements compared to previous works. Figure 3 shows the original proposed algorithm in [13].

```
crc = INIT_VALUE;
while(p_buf<p_end ) {
crc ^= *(uint32_t *)p_buf;
  term1 = table_56[crc& 0x000000FF] ^
    table_48[(crc>> 8) & 0x000000FF];
  term2 = crc>> 16;
crc = term1 ^ table_40[term2 &
    0x000000FF] ^ table_32[(term2 >> 8)
& 0x000000FF];
p_buf += 4;
}
returncrc^FINAL_VALUE;
```

Algorithm 3: Slicing by 4 from [13]

D. CRC in operating systems

1) Linux

The Linux operating system is the most popular open source operating system for all embedded, mobile, server and desktop usages. At the Linux kernel library directory, there are many implementations for CRC including CRC-7, CRC-8, CRC16, CRC-CCITT, CRC-ITU-T and during compile time, the main algorithm of CRC-32 can be chosen to one of the following algorithms:

1. Slice-By-8 (the default) [13]
2. Slice-By-4 [13]
3. Sarwate's Algorithm [5](one byte at a time)
4. Classic one Algorithm (one bit at the time)

Algorithm 4 is located in "/lib/xz/xz_crc32.c" and is used in XZ decompression.

```
XZ_EXTERN uint32_t
 xz_crc32(const uint8_t *buf,
size_t size, uint32_t crc)
{
crc = ~crc;
while (size != 0) {
crc = xz_crc32_table[*buf++ ^
    (crc& 0xFF)] ^ (crc>> 8);
 --size;
 }
return ~crc;
}
```

Algorithm 4: Part of Linux Kernel (CRC-32)

2) FreeBSD

The FreeBSD operating system is one of the oldest operating systems with a friendly license for commercial use. Algorithm 5 is the CRC-32 function within the kernel source code (sys/libkern/crc32.c). You can also find implementations for CRC-32C and slicing-by-8 within the same file. This algorithm is using "crc32_tab" as a predefined look-up table for CRC computations.

```
uint32_t
crc32(const void *buf, size_t size)
{
const uint8_t *p = buf;
uint32_tcrc;
crc = ~0U;
while (size--)
crc = crc32_tab[(crc ^ *p++) &
        0xFF] ^ (crc>> 8);
returncrc ^ ~0U;
}
```

Algorithm 5: Part of FreeBSD Kernel (CRC-32)

IV. Conclusion

The aim of this paper was to provide a brief review for cyclic redundancy check method known as CRC mechanism applied to some popular FOSS applications. Based on the works of [11, 8, 9, 13] there are still many investigation research areas for improving these mechanisms. The main efforts are directed to improve the characteristics of CRC by finding better implementations and more efficient polynomial generators to improve CRC efficiency especially for large amounts of data transfers in FOSS and other applications.

Acknowledgments

This research was carried out at the Free & Open Source Software (FOSS) Lab, under the Communication & Information Research Center (CIRC) at Sultan Qaboos University (SQU) - Oman, (www.squ.edu.om/circ/). The researchers would like to acknowledge the support from Information Technology Authority, ITA, for providing the needed resources.

References

[1] Andrew S. Tanenbaum, David J. Wetherall, Computer Networks. Prentice Hall, 2010.
[2] W W Peterson, D T Brown, “Cyclic Codes for Error Detection”, Proceedings of the IRE, vol. 49, no. 1, pp. 228—235, 1961.
[3] Aram Perez, “Byte-Wise CRC Calculations”, IEEE Micro, vol. 3, no. 3, pp. 40—50, 1983.
[4] T V Ramabadran, S SGaitonde, “A tutorial on CRC computations”, Micro, IEEE, vol. 8, no. 4, pp. 62—75, 1988.
[5] D V Sarwate, “Computation of Cyclic Redundancy Checks via Table Look-up”, Commun. ACM, vol. 31, no. 8, pp. 1008—1013, 1988.
[6] Guido Albertengo, Riccardo Sisto, “Parallel CRC Generation”, IEEE Micro, vol. 10, no. 5, pp. 63—71, 1990.
[7] G Castagnoli, S Brauer, M Herrmann, “Optimization of cyclic redundancy-check codes with 24 and 32 parity bits”, Communications, IEEE Transactions on, vol. 41, no. 6, pp. 883—892, 1993.
[8] P Koopman, “32-bit cyclic redundancy codes for Internet applications”. 2002.
[9] P Koopman, T Chakravarty, “Cyclic redundancy code (CRC) polynomial selection for embedded networks”. 2004.
[10] Vinodh Gopal, ErdincOzturk, James Guilford, WajdiFeghali, “Choosing a CRC polynomial and associated method for Fast CRC Computation on Intel Processors”. 2012. URL http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/fast-crc-computation-paper.pdf.
[11]A Ahmad, L Hayat, “Selection of Polynomials for Cyclic Redundancy Check for the Use of High Speed Embedded: An Algorithmic Procedure”, W. Trans. on Comp., vol. 10, no. 1, pp. 16—20, 2011. URL http://dl.acm.org/citation.cfm?id=2001170.2001173.
[12] Michael Barr, “Slow and steady never lost the race”, Embedded Systems Programming, vol. , no. , pp. 37—46, 2000.URL http://www.barrgroup.com/Embedded-Systems/How-To/CRC-Calculation-C-Code.
[13] Michael E Kounavis, Frank L Berry, “A Systematic Approach to Building High Performance Software-Based CRC Generators”, 2013 IEEE Symposium on Computers and Communications (ISCC), vol. 0, no. , pp. 855—862, 2005.
[14] Cypress Semiconductor Corporation, “Cyclic Redundancy Check(CRC),” 2013.

Some common CRC polynomials

The following list includes some of the common CRC polynomials and what they are known for. [14]

- CRC-1: The simplest form of CRC for parity checking. Polynomial is:$x + 1$
- CRC-4-ITU Used in ITU G.704, the polynomial is: $x^4 + x + 1$
- CRC-5-ITU Used in ITU G.704, the polynomial is: $x^5 + x^4 + x^2 + 1$
- CRC-5-USB Used in USB, the polynomial is: $x^5 + x^2 + 1$
- CRC-6-ITU Used in ITU G.704, the polynomial is: $x^6 + x + 1$
- CRC-7 Used in Telecom systems and Multimedia Card (MMC). Polynomial is:$x^7 + x^3 + 1$
- CRC-8 General, Polynomial is: $x^8 + x^7 + x^6 + x^4 + x^2 + 1$
- CRC-8-ATM Used in ATM HEC, Polynomial is: $x^8 + x^2 + x + 1$
- CRC-8-CCITT Used in 1-Wire bus, Polynomial is: $x^8 + x^7 + x^3 + x^2 + 1$
- CRC-8-Maxim Used in 1-Wire bus, Polynomial is: $x^8 + x^5 + x^4 + 1$
- CRC-8-SAE Used in SAE J1850, Polynomial is: $x^8 + x^4 + x^3 + x^2 + 1$
- CRC-10 General, Polynomial is: $x^{10} + x^9 + x^5 + x^4 + x + 1$
- CRC-12 Used in Telecom systems, Polynomial is: $x^{12} + x^{11} + x^3 + x^2 + x + 1$
- CRC-15-CAN Used in CAN, Polynomial is: $x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$
- CRC-16 Used in USB. Polynomial is: $x^{16} + x^{15} + x^2 + 1$
- CRC-16-CCITT Also known as CRC-CCITT and CRC-IBM, used in HDLC, SDLC, XMODEM, X.25, V.41, Bluetooth, PPP, IrDA, Polynomial is: $x^{16} + x^{12} + x^5 + 1$
- CRC-24-Radix64 General, Polynomial is: $x^{24} + x^{23} + x^{18} + x^{17} + x^{14} + x^{11} + x^{10} + x^7 + x^6 + x^5 + x^4 + x^3 + x + 1$
- CRC-32 Also known as CRC-IEEE, used in Ethernet, AAL5 (ATM Adaptation Layer 5) , FDDI (Fiber Distributed Data Interface) . Polynomial is: $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$
- CRC-32CKnow as CRC-32C or Castagnoli, used in iSCSI.Polynomial is: $x^{32} + x^{28} + x^{27} + x^{26} + x^{25} + x^{23} + x^{22} + x^{20} + x^{19} + x^{18} + x^{14} + x^{13} + x^{11} + x^{10} + x^9 + x^8 + x^6 + 1$
- CRC-32KKnow as CRC-32K or Koopman, Polynomial is: $x^{32} + x^{30} + x^{29} + x^{28} + x^{26} + x^{20} + x^{19} + x^{17} + x^{16} + x^{15} + x^{11} + x^{10} + x^7 + x^6 + x^4 + x^2 + x + 1$
- CRC-64-ISO Defined in ISO 3309, Polynomial is: $x^{64} + x^4 + x^3 + x + 1$