PERANCANGAN DAN IMPLEMENTASI *CYCLIC REDUNDANCY CHECK* – 16 SEBAGAI METODE *ERROR CHECKING* PADA TRANSMISI PESAN PROTOKOL MODBUS *REMOTE TERMINAL UNIT* BERBASIS *MICROCONTROLLER UNIT*

Arief Wisnu Wardhana, Eka Firmansyah, Addin Suwastono
Jurusan Teknik Elektro, Fakultas Teknik, Universitas Gadjah Mada
Jalan Grafika 2 Yogyakarta 55281 Indonesia
*Corresponding author, e-mail : awwardhana.sie13@mail.ugm.ac.id

Abstrak—Paper ini membahas tentang Cyclic Redundancy Check – 16, sebuah generator polynomial untuk mendeteksi error, yang biasanya digunakan pada MODBUS Remote Terminal Unit. Diawali dengan penjelasan tentang fenomena derau yang biasanya menyertai sebuah sinyal utama ketika sinyal tersebut ditransmisikan melalui sebuah kanal berderau. Beberapa tipe error yang biasanya mempengaruhi bit bit dari byte data yang ditransmisikan kemudian dijelaskan. Metode lengkap untuk mendesain sebuah generator polynomial untuk mendeteksi error kemudian dipaparkan. Selanjutnya, Cyclic Redundancy Check – 16 sebagai sebuah contoh generator polynomial dibahas. Pembahasan meliputi metode untuk implemenrasi software dari CRC tersebut. Dua metode diperkenalkan yaitu metode loop – driven dan metode table – driven. Pada bagian akhir, ditunjukkan hasil dari generator polynomial yang dirancang, yang terdiri dari algoritma nya dan salah satu contoh rutin programnya. Rutin CRC – 16 tersebut kemudian dites dengan menggunakan beberapa pesan MODBUS.

Kata Kunci - Noise, generator polynomial, Cyclic Redundancy Check, MODBUS Remote Terminal Unit, lood – driven, table – driven, algorithm

Abstract—This paper presents about the Cyclic Redundancy Check – 16, a generator polynomial for error detection, which is normally used in MODBUS Remote Terminal Unit. It starts with explaining the noise phenomena that are often generated into a useful signal when it is transmitted through a noisy channel. Some types of error which usually affects bits of a transmitted data bytes are introduced. The complete method for designing a good generator polynomial for detecting the error is then presented. Next, Cyclic Redundancy Check -16 as an example of generator polynimial is discussed. The explanation includes method for software implementation of the CRC. Two methods are introduced which are loop – driven method and table - driven method. Finally, result of designed generator polynomial is shown, which consists of the algorithm and the routine example. The CRC -16 routine is then tested using some MODBUS messages.

Keywords - Noise, generator polynomial, Cyclic Redundancy Check, MODBUS Remote Terminal Unit, lood – driven, table – driven, algorithm

1. PENDAHULUAN

Jaringan komunikasi *transmit receive* harus bisa mentransfer data dari satu piranti ke piranti yang satunya lagi. Kapanpun data ditransmisikan dari satu node ke node berikutnya, mereka bisa menjadi terkorupsi dalam perjalanannya. Banyak faktor bisa mengubah satu atau lebih bit dari sebuah *message*. Beberapa aplikasi memerlukan sebuah mekanisme untuk mendeteksi dan mengkoreksi *error*.

Beberapa aplikasi bisa mentoleransi sebuah *error* level kecil. Misalnya, *random error* dalam transmisi audio atau video mungkin bisa ditoleransi, tetapi ketika kita mentransfer text, kita mengharapkan sebuah level akurasi yang sangat tinggi [2]. Pen transfer an text salah satunya misalnya dilakukan pada protokol komunikasi MODBUS *Remote Terminal Unit*. Pada protokol tersebut terjadi pengiriman text yaitu berupa pengiriman *message* dari *transmiter* ke *receiver*. Text tersebut berupa kumpulan byte 8-bit. Tiap tiap byte 8-bit dalam *message*

Received date 2016-01-27, Revised date 2016-03-08, Accepted date 2016-03-10
DOI : 10.20449/jnte.v5i1.189

tersebut terdiri dari dua buah karakter (huruf) 4 - bit hexadesimal (hexadesimal artinya bernilai 0 sampai F). MODBUS RTU ini banyak digunakan pada aplikasi kontrol. Rofan Aziz dan Karsid [7] membahas uji performansi kontrol suhu dan kelembaban menggunakan variasi kontrol digital dan kontrol scheduling untuk pengawetan buah dan sayuran.

Permasalahan utama yang akan diangkat di sini adalah menyediakan sebuah cara pendeteksian error yang paling sesuai untuk digunakan pada transmisi byte – byte data di protokol komunikasi MODBUS Remote Terminal Unit.

Untuk deteksi dan koreksi error tersedia *Block Code, Hammimg Code*. Sesuai dengan persoalan yang ingin diselesaikan, paper ini membahas tenntang mekanisme pendeteksian error menggunakan CRC bit.

CRC digunakan untuk menjamin integritas paket data yang ditransfer antar dua *microcontroller unit* atau antar *microcontroller unit* dan komputer melalui sebuah interface seperti UART / USART, SPI, I2C dan sebagainya. Beberapa MCU terkini mempunyai fasilitas *hardware generation* untuk CRC untuk USART, I2C. Apa yang dilakukan di paper ini adalah mengimplementasikan CRC secara *software*.

Paper ini disusun dalam tiga bagian utama. Bagian pertama menjelaskan tentang bit *error* pada transmisi data. Bagian kedua menjelaskan perancangan *generator polynomial* secara umum yang kemudian mengarah pada perancangan *generator polynomial* CRC – 16. Bagian ketiga menjelaskan tentang implementasi CRC – 16 yang sudah dibuat.

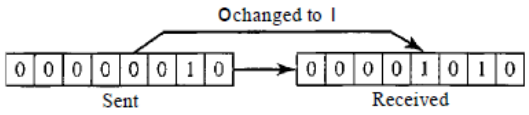
2. ERROR PADA TRANSMISI DATA [2]

2.1. Tipe – Tipe Bit Error

2.1.1. Single-Bit Error

Istilah *single-bit-error* berarti bahwa hanya 1 bit dari unit data yang diberikan (seperti misalnya sebuah byte, karakter, atau paket) berubah dari 1 ke 0 atau dari 0 ke 1. Pada papernya, Sunil Shukla, Neil W. Bergmann [8] membahas tentang implementasi single – bit error ini.

Gambar 1. menunjukkan efek dari sebuah *single – bit error* pada sebuah unit data



Gambar 1. Single – bit error

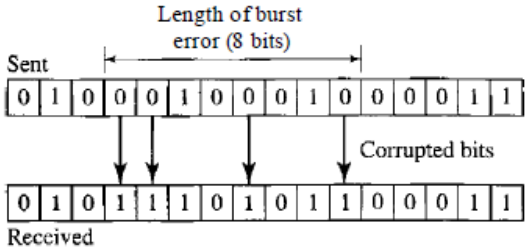
Untuk memahami akibat dari perubahan tersebut, bisa dijelaskan sebagai berikut. Tiap tiap grup sebesar 8 bit adalah sebuah karakter ASCII dengan bit 0 ditambahkan di sebelah kirinya. Pada gambar di atas, 00000010 (ASCII STX) dikirimkan, yang berarti *start of text*, tetapi, 00001010 diterima, yang berarti *line feed*.

Single – bit error adalah *error* yang paling kecil kemungkinannya terjadi dalam transmisi data serial. Data terkirim pada 1 Mbps. Ini berarti bahwa setiap bit berlangsung hanya 1/1000000 s atau 1µs. Agar sebuah *single - bit error* terjadi, *noise* harus memepunyai sebuah durasi hanya 1 µs, yang adalah sangat jarang; *noise* biasanya berlangsung jauh lebih lama dari ini.

2.1.2. Burst Error

Istilah *burst error* berarti bahwa 2 atau lebih bit pada unit data telah berubah dari 1 ke 0 atau dari 0 ke 1. Yoshihisa Desaki et.al [10] membahas tentang mekanisme double bit dan triple bit error.

Gambar 2. menunjukkan efek dari sebuah *burst error* pada sebuah unit data.



Gambar 2. Burst error dengan panjang burst 8

Pada kasus ini 0100010001000011 dikirimkan, tetapi 0101110101100011 yang diterima. Perhatikan bahwa sebuah *burst error* tidak berarti *error* harus terjadi pada bit secara berturutan. Panjang *burst* diukur dari bit pertama yang terkorupsi sampai bit terakhir yang terkorupsi. Beberapa bit diantara kedua bit tersebut bisa saja tidak terkorupsi.

Sebuah *burst error* lebih mungkin untuk terjadi dibandingkan sebuah *single – bit error*.

Durasi *noise* biasanya lebih panjang daripada durasi 1 bit, yang berarti bahwa ketika *noise* mempengaruhi data, dia akan mempengaruhi satu set bit (lebih dari satu bit). Jumlah bit yang terkena efeknya tergantung pada *data rate* dan durasi *noise* itu. Sebagai contoh, jika kita mengirim data pada 1 Kbps, sebuah *noise* 11100 s bisa mempengaruhi 10 (sepuluh) bit; jika kita mengirim data pada 1 Mbps, *noise* yang sama bisa mempengaruhi 10000 (sepuluh ribu) bit.

2.2. Redundancy

Konsep utama dalam pendeteksian atau pengkoreksian *error* adalah *redundancy*. Agar bisa mendeteksi atau mengkoreksi *error*, kita perlu untuk mengirimkan bit bit extra bersamaan dengan data kita. Bit bit *redundant* ini ditambahkan oleh pengirim dan dibuang oleh *receiver*. Keberadaan nya memungkinkan *receiver* untuk mendeteksi atau mengkoreksi bit terkorupsi.

2.3. Deteksi versus Koreksi

Koreksi *error* adalah lebih sulit daripada deteksi. Pada deteksi *error*, kita mencari hanya untuk melihat apakah *error* telah terjadi. Jawabannya adalah hanya *ya* atau *tidak*. Kita bahkan tidak tertarik pada berapa jumlah *error* nya. Sebuah *single – bit error* adalah sama bagi kita seperti sebuah *burst error*.

Pada koreksi *error*, kita perlu untuk mengetahui jumlah tepatnya bit yang terkorupsi dan lebih penting lagi, lokasi bit tersebut di dalam *message*. Jumlah *error* dan ukuran *message* adalah faktor penting. Apabila kita ingin untuk mengkoreksi satu *single – error* di dalam sebuah unit data 8 – bit, kita perlu untuk mempertimbangkan delapan kemungkinan lokasi *error*; jika kita ingin mengkoreksi dua *error* dalam sebuah unit data dengan ukuran yang sama, kita perlu untuk mempertimbangkan 28 kemungkinan.

2.4. Forward Error Corection versus Re-transmission

Ada dua metode utama unuk koreksi *error*. Koreksi *forward error* adalah proses dimana *receiver* berusaha untuk memperkirakan *message* dengan menggunakan *redundant* bit. Ini dimungkinkan, apabila jumlah *error* kecil.

Koreksi dengan re-transmisi adalah sebuah teknik dimana *receiver* mendeteksi adanya sebuah *error* dan meminta *sender* untuk mengirimkan kembali *message* nya. Pengiriman kembali *message* diulang sampai sebuah *message* tiba dimana *receiver* percaya *message* tersebut adalah *error – free* (biasanya tidak semua *error* bisa dideteksi).

Tergantung dari sistem komunikasi nya, kita bisa saja menggunakan sebuah skema *error detection* (yang bisa meminta *re - transmission* data dari *transmitter*) atau skema *error – correction*, atau kedua duanya skema *error – detection* dan *error correction*.

CRC – 16 merupakan skema *error detection* (yang bisa meminta *re - transmission* data dari *transmitter*).

3. GENERATOR POLYNOMIAL DAN CRC – 16 [2][3][6]

3.1. Generator Polynomial [2][3]

Parity, atau *block sum check* turunan nya, tidak menyediakan sebuah skema pendeteksian yang handal terhadap *error*. Pada kasus seperti itu, alternatif yang paling umum adalah berbasis pada penggunaan *polynomial codes*. *Polynomial codes* digunakan dengan skema transmisi *frame* (blok). Jonathan Stone et. al [6] dalam paparannya membandingkan performansi antara checksum dan CRC pada *real data*.

Satu set digit digit pengecekan dihasilkan (dihitung) untuk setiap *frame* yang ditransmisikan, berdasar pada isi dari frame tersebut, dan kemudian ditambahkan oleh *transmitter* pada akhir dari *frame*. *Receiver* kemudian melakukan sebuah perhitungan yang serupa pada *frame* dan *check digit*. Apabila tidak terjadi *error*, maka hasil yang sudah diketahui diperoleh. Apabila hasilnya berbeda, mengindikasikan kesalahan. Oleh H. Michael Ji dan Earl Killian [5], dijelaskan tentang alghoritma dan implementasi *Fast Parallel* CRC pada sebuah *Configurable Processor*.

Jumlah *check digit* untuk setiap frame dipilih untuk menyesuaikan dengan tipe *error* transmisi yang diantisipasi. Pada prakteknya, digunakan 16, 24, or 32-bit *generator vectors* untuk menghasilkan bit bit *check digit* untuk menaikkan secara signifikan *error detection rate*, meskipun 16 dan 32 bit adalah yang paling umum. Hasil *check digit* yang dihitung dinamakan *Frame Check Sequence* (FCS) atau *Cyclic Redundancy Check* (CRC) [3]. Dinamakan kode *Cyclic Redundancy Check* (CRCs) karena *Check code* (verifikasi data) nya adalah sebuah

Redundancy (tidak memberikan informasi apapun) dan algorithma pembuatan nya berbasis pada *Cyclic codes*, Wael M El-Medany [8], seperti yang akan dijelaskan pada bagian selanjutnya.

Metode perhitungan CRC menggunakan properti *binary numbers* jika digunakan modulo – 2 arithmetics. Misalkan :

M(x) adalah sebuah bilangan k-bit (*message* yang ditransmisikan)
G(x) adalah sebuah bilangan (n+1)-bit (pembagi atau *generator*)
R(x) adalah sebuah bilangan n-bit sedemikian rupa sehingga k > n (sisa)

Maka, jika :

$$\frac{M(x)\ x\ 2^n}{G(x)} = Q(x) + \frac{R(x)}{G(x)} \qquad (1)$$

Dimana Q(x) adalah *quotient* (hasil)

$$\frac{M(x)\ x\ 2^n + R(x)}{G(x)} = Q(x) \qquad (2)$$

dengan mengasumsikan aritmetika modulo-2.

Kita bisa memastikan hasilnya dengan mengganti ekspresi untuk $\frac{M(x)\ x\ 2^n}{G(x)}$ ke dalam persamaan kedua, memberikan :

$$\frac{M(x)\ x\ 2^n + R(x)}{G(x)} = Q(x) + \frac{R(x)}{G(x)} + \frac{R(x)}{G(x)} \qquad (3)$$

yang hasilnya adalah sama dengan *Q(x)* karena bilangan apapun ditambahkan ke dirinya sendiri modulo-2 akan menghasilkan nol, artinya, sisanya adalah nol.

Isi *frame* lengkap, M(x), bersama dengan satu set nol tambahan dengan jumlah nol sama dengan jumlah digit FCS (CRC) yang akan dibuat (yang adalah ekuivalen dengan mengalikan *message* tersebut dengan 2^n, dimana *n* adalah jumlah digit CRC) dibagi *modulo-2* dengan sebuah bilangan biner kedua, G(x), dinamakan ***generator polynomial***, yang berisi satu digit lebih banyak daripada CRC. Operasi pembagian ekuivalen dengan melakukan operasi exclusive-OR bit demi bit secara parallel seiring setiap bit pada *frame* tersebut diproses [4].

XOR	0	1
0	0	1
1	1	0

AND	0	1
0	0	0
1	0	1

Gambar 3. Tabel modulo-2 XOR dan modulo-2 AND

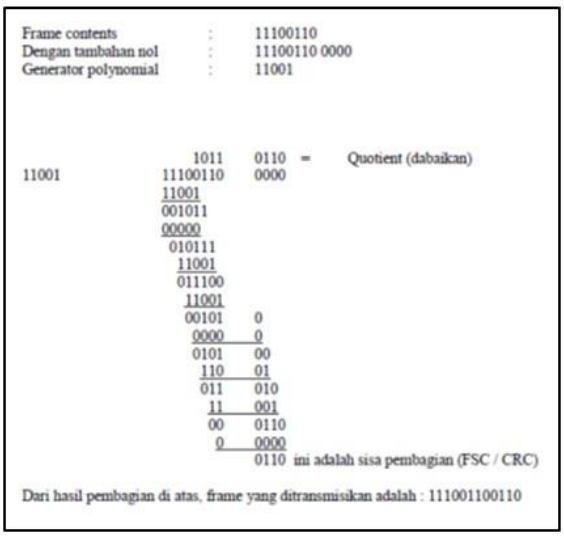
Sisanya *R(x)* adalah CRC yang ditransmisikan pada bagian akhir dari digit informasi. Secara sama, pada bagian penerimaan, aliran bit yang diterima termasuk CRC dibagi lagi dengan *generator polynomial* yang sama – jadi, $\frac{M(x)\ x\ 2^n + R(x)}{G(x)}$ - dan, jika tidak ada *error* terjadi, sisanya semua nol. Jika sebuah *error* terjadi, sisanya adalah tidak nol.

Selanjutnya, kita akan mendiskusikan sebuah contoh acak perhitungan bit bit CRC, setelah diberikan contoh bit bit data pesan dan *generator polynomial*. Kita merepresentasikan semua input dan output dari modul CRC, seperti misalnya data pesan, CRC *generator*, dan nilai CRC itu sendiri, dinyatakan dalam bit dan pada akhirnya dinyatakan dalam *polynomial* pada perhitungan bit bit CRC. Misalnya, sebuah *binary vector* ***b*** = [11100110] direpresentasikan dalam bentuk *polynomial* sebagai berikut

$$b(x) = 1*x7 + 1*x6 + 1*x5 + 0*x4 + 0*x3 + 1*x2 + 1*x + 0$$
$$= x7 + x6 + x5 + x2 + x1 \qquad (4)$$

Data pesan ***b*** = [11100110] adalah *dividend* dan ***g*** = [11001] adalah *divisor* dari operasi pembagian. Dalam notasi *polynomial*, ekuivalennya direpresentasikan sebagai *b(x) = x7 + x6 + x5 + x2 + x1* dan *g(x) = x4 + x3 + 1*. Sisa pembagian diperoleh dalam bentuk vektor sebagai **c** = [0110] atau dalam bentuk *polynomial* sebagai *c(x) = x2 + x 1*.

Dengan kata lain, apabila ***b*** = [11100110] adalah *message* dan ***g*** = [11001] adalah sebuah *generator polynomial*, maka sisa ***c*** = [0110] adalah nilai CRC nya. Kita menambahkan bit bit CRC ke *message* data sebagai **m** = b | c dan mentransmisikan nya ke *receiver*.

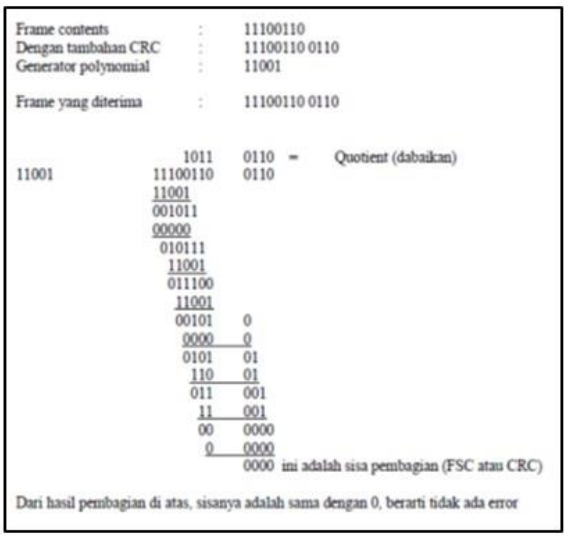


Gambar 4. Proses menghasilkan CRC

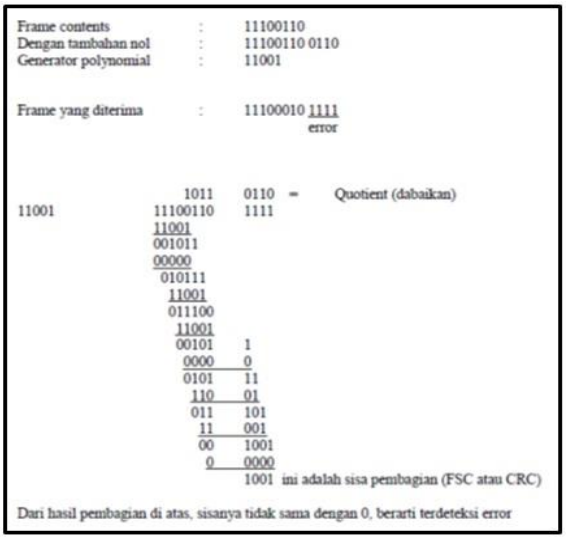
Pada gambar 4 terlihat bahwa pertama tama empat buah nol ditambahkan ke *message*, yang ekuivalen dengan mengalikan *message* tersebut dengan 2^4, karena CRC akan terdiri dari empat bit (*generator polynomial* berisi satu digit lebih banyak daripada CRC). Ini kemudian dibagi (modulo-2) dengan *generator polynomial* tersebut. Operasi pembagian modulo-2 ekuivalen dengan melakukan operasi exclusive-OR bit demi bit secara parallel seiring setiap bit pada *dividend* diproses. Juga, dengan aritmetika modulo-2, kita bisa melakukan sebuah pembagian terhadap setiap sisa parsialnya, asalkan kedua bilangan mempunyai panjang yang sama, dalam artian, *most significant bit* keduanya adalah 1. Kita mengabaikan besar relatif dari kedua bilangan tersebut. Hasil sisa 4-bit yang dihasilkan (0110) adalah FCS atau CRC. Sementara *quotient* tidak digunakan.

Pada sisi *receiver*, keseluruhan sekuensi bit yang diterima dibagi dengan *generator polynomial* yang sama seperti yang digunakan pada *transmitter*. Dua buah contoh diberikan di bawah ini. Pada contoh pertama (Gambar 5), diasumsikan tidak ada *error* terjadi selama transmisi, sehingga sisa pembagian adalah nol – *quotient* tidak digunakan.

Pada contoh kedua (Gambar 6), diasumsikan terjadi sebuah *error burst* sebanyak empat bit pada akhir sekuensi bit yang ditransmisikan. Konsekuensinya, sisa pembagian yang dihasilkan tidak nol, mengindikasikan bahwa sebuah *error* transmisi telah terjadi.



Gambar 5. Proses pengecekan – tidak ada error



Gambar 6. CRC checking process – terdeteksi error

Pemilihan *generator polynomial* menjadi penting karena akan menentukan tipe *error* yang akan dideteksi. Misalnya, *frame* yang ditransmisikan, *T(x)* adalah 110101100110, dan *error pattern* (sebuah sekuensi dengan panjang – N dengan angka angka 1 pada posisi dimana pembalikan bit atau *error* terjadi) *E(x)* adalah 000000001001. Pola *error* tersebut mengindikasikan bahwa *error* telah terjadi pada posisi bit dengan angka 1. Dengan modulo-2 arithmetic, *frame* yang diterima adalah

$$T(x) + E(x) \qquad (5)$$

$$\frac{T(x)+E(x)}{G(x)} = \frac{T(x)}{G(x)} + \frac{E(x)}{G(x)} \quad (6)$$

Pembagian *T(x) / G(x)* tidak menghasilkan sisa. Sisa yang kita peroleh ketika membagi dengan *G(x)* hanyalah sisa yang didapatkan dari pembagian *E(x)* dengan *G(x)*. Jadi, *error* hanya akan terdseteksi hanya jika *E(x) / G(x)* menghasilkan sebuah sisa.

Misalnya, semua G(x) mempunyai paling sedikit tiga term (bit bit 1) dan *E(x) / G(x)* akan menghasilkan sebuah sisa untuk semua error *single – bit* dan semua *error double – bit* dengan modulo-2 arithmetic dan oleh karena nya *error* terdeteksi. Sebaliknya, sebuah *error burst* dengan panjang yang sama dengan *G(x)* bisa saja merupakan sebuah kelipatan *G(x)* dan oleh karenanya tidak menghasilkan sisa dan *error* tidak terdeteksi.

Bisa ditunjukkan (dengan menganggap bahwa *E(x) / G(x)* akan mempunyai sebuah sisa) bahwa dengan sebuah FCS (atau CRC) n – bit, kita bisa mendesain *generator polynomial* sedemikian rupa sehingga mungkin untuk mendeteksi *single bit error*, dua *single – bit error* yang terisolasi, sejumlah ganjil *error*, dan *burst error*.

Dengan melihat kepada beberapa *error* spesifik seperti yang ditunjukkan di sub bab tipe tipe *error* di atas, kita bisa melihat bagaimana *error* tersebut bisa terdeteksi dengan sebuah *polynomial generator g(x)* yang terdesain secara baik.

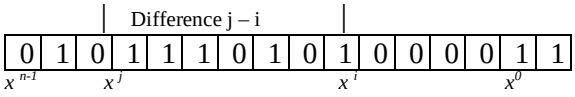
Generator Polynomial untuk Single-Bit Error

Bagaimana seharusnya struktur *g(x)* untuk menjamin terdeteksinya sebuah *single – bit error* ? Sebuah *single – bit error* adalah *E(x) = xi*, dimana *i* adalah posisi dari *error bit* tersebut. Jika sebuah *single – bit error* terdeteksi, maka *xi* tidak bisa terbagi dengan *g(x)*. (Perhatikan bahwa ketika kita mengatakan *tidak bisa terbagi*, maksudnya adalah ada sebuah sisa). Jika *g(x)* mempunyai paling sedikit dua terms (seperti yang pada umumnya terjadi) dan koefisien *x0* nya tidak nol (bit paling kanan adalah 1), maka *e(x)* tidak bisa dibagi dengan *g(x)* [2].

Jadi, jika *generator polynomial* mempunyai lebih dari satu term dan koefisien dari *x0* adalah 1, maka semua *single - bit error* terdeteksi.

Generator Polynomial untuk two Isolated Single-Bit Errors

Jenis kedua adalah ada dua buah *single – bit error* yang terisolasi satu sama lain. Dengan kondisi bagaimana tipe *error* ini bisa terdeteksi ? Kita bisa menunjukkan tipe *error* ini sebagai *e(x) = xj + xi*. Nilai *i* dan *j* mendefinisikan posisi dari *error*, dan perbedaan *j – i* mendefinisikan jarak antara kedua *error* tersebut, seperti yang ditunjukkan di Gambar 7.



Gambar 7. Representasi dari dua single- bit error yang terisolasi menggunakan polynomial

Kita bisa menulis $e(x) = x^i (x^{j-i} + 1)$. Jika g(x) mempunyai lebih dari satu term dan satu term adalah *x0*, *g(x)* tidak bisa membagi x^i, seperti yang kita lihat sub pada bagian sebelumnya. Jadi agar *g(x)* bisa membagi *e(x)*, *g(x)* harus bisa membagi $x^{j-i} + 1$. Dengan kata lain, *g(x)* harus tidak bisa membagi $x^t + 1$, dimana *t* adalah antara 0 dan *n-1*. Tetapi, $t = 0$ tidak ada artinya dan $t = 1$ diperlukan seperti yang akan kita lihat nanti. Ini berarti *t* harus berada di antara 2 dan *n-1*.

Jadi, jika sebuah *generator polynomial* tidak bisa membagi $x^t + 1$ (*t* antara 0 dan *n-1*), maka semua *isolated double error* bisa terdeteksi.

Generator Polynomial untuk Odd Numbers of Errors

Sebuah *generator polynomial* dengan sebuah faktor x + 1 bisa menangkap semua *error* berjumlah ganjil. Ini berarti bahwa kita perlu untuk membuat x + 1 sebuah faktor dari *generator polynomial* apapun. Perhatikan bahwa kita tidak mengatakan bahwa *generator polynomial* itu sendiri yang harus x + 1; kita mengatakan bahwa *generator polynomial* tersebut harus mempunyai sebuah faktor x + 1. Jika dia hanya x + 1, dia tidak bisa mendeteksi dua *error* terpisah yang berdampingan (lihat sub bagian sebelumnya)

Sebagai contoh, *x4 + x2 + x + 1* bisa menangkap semua *odd – number error* karena dia ditulis sebagai sebuah hasil perkalian dari dua *polynomial* yaitu x + 1 dan x3 + x2 + 1.

Jadi, sebuah *generator polynomial* yang berisi faktor x + 1 bisa mendeteksi semua *odd - number error*.

Generator Polynomial untuk Burst Errors (2 atau lebih bit berubah)

Sekarang mari kia perluas analisis kita ke *burst error*, yang adalah paling penting dari semuanya. Sebuah *burst error* adalah berbentuk $e(x) = (x^j + ... + x^i)$. Perhatikan perbedaan antara sebuah *burst error* dengan dua *isolated single – bit error*. Yang pertama bisa mempunyai dua term atau lebih; yang kedua hanya mempunyai dua term. Kita bisa memfaktorkan xi dan menuliskan *error* sebagai $x^i (x^{j-i} + ... + 1)$. Jika *generator polynomial* kita bisa merndeteksi sebuah single *error* (sebuah kondisi minimum untuk sebuah *generator polynomial*), maka dia tidak bisa membagi xi. Apa yang harus kita khawatirkan tentang *generator polynomial* yang bisa membagi $x^{j-i} + ... + 1$. Dengan kata lain, sisa dari $(x^{j-i} + ... + 1) / (x^r + ... + 1)$ tidak nol. Perhatikan bahwa denominator adalah *generator polynomial*. Kita bisa mempunyai tiga kasus :

1. Jika $j – i < r$, sisanya tidak akan pernah nol. Kita bisa menulis $j – i = L – 1$, dimana L adalah panjang *error*. Jadi, $L -1 < r$ atau $L < r + 1$ atau $L \leq r$. Ini berarti semua *burts error* dengan panjang lebih kecil atau sama dengan jumlah *check* bit r akan terdeteksi.
2. Dalam beberapa kasus yang jarang terjadi, jika $j – i = r$, atau $L = r + 1$, *syndrome* (sisa) adalah nol dan *error* nya tak terdeteksi. Bisa dibuktikan bahwa pada kasus ini, probabilitas *burst error* dengan panjang $r + 1$ tak terdeteksi adalah $(1/2)^{r-1}$. Sebagai contoh, jika *polynomial generator* kita adalah $x^{14} + x^3 + 1$, dimana $r = 14$, sebuah *burst error* dengan panjang $L = 15$ bisa tak terdeteksi dengan probabilitas $(1/2)^{14-1}$ atau hampir 1 dalam 10000.
3. Dalam beberapa kasus yang jarang terjadi, jika $j – i > r$, atau $L > r + 1$, *syndrome* nya adalah nol dan *error* nya tak terdeteksi. Bisa dibuktikan bahwa pada kasus ini, probabilitas *burst error* dengan panjang lebih besar daripada $r + 1$ tak terdeteksi adalah $(1/2)^r$. Sebagai contoh, jika *polynomial generator* kita adalah $x^{14} + x^3 + 1$, dimana $r = 14$, sebuah *burst error* dengan panjang lebih besar daripada $L = 15$ bisa tak terdeteksi dengan probabilitas $(1/2)^{14}$ atau hampir 1 dalam 16000 kasus

Kesimpulannya, sebuah *generator polynomial* sebesar R bit akan bisa mendeteksi :

- Semua *single – bit error*
- Semua *double - bit error*
- Semua *odd number* (sejumlah ganjil) *error*
- Semua *burst error* < R
- Kebanyakan *burst error* ≥ R

Atau dengan kata lain, sebuah *generator polynomial* yang baik perlu untuk memiliki karakteristik berikut ini :

1. *Generator polynomial* harus mempunyai paling sedikit dua term.
2. Koefisien dari term x0 harus 1
3. *Generator polynomial* harus tidak bisa membagi $x^t + 1$, untuk t antara 2 dan n – 1.
4. *Generator polynomial* harus punya faktor $x + 1$

Tabel 1. Beberapa standar generator polynomial yang digunakan oleh berbagai protokol

Nama	Polynomial Generator	Aplikasi
CRC – 8	$x^8 + x^2 + x + 1$	ATM Header
CRC – 10	$x^{10} + x^9 + x^5 + x^4 + x^2 + 1$	ATMAAL
CRC - 16	$x^{16} + x^{15} + x^2 + 1$	HDLC, WANs, MODBUS
CRC – CCITT	$x^{16} + x^{15} + x^5 + 1$	WANs
CRC – 32	$x^{32} + x^{26} + x^{23} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$	LANs

Cara standar untuk merepresentasikan sebuah *generator polynomial* adalah menunjukkan posisi yang merupakan binary 1 sebagai pangkat x. Beberapa standar *generator polynomial* yang digunakan oleh berbagai protokol untuk menghasilkan CRC ditunjukkan di Tabel 1.

3.2. Generator Polynomial untuk CRC- 16 [1]

Generator polynomial untuk CRC-16 ditunjukkan pada persamaan 7.

$$P(x) = x^{16} + x^{15} + x^2 + 1 \quad (7)$$

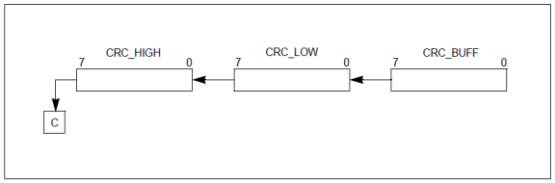
Polynomial di atas diterjemahkan ke dalam sebuah nilai *binary* karena divisor dilihat sebagai sebuah *polynomial* dengan koefisien *binary*.

Jadi, *generator polynomial* CRC-16 diterjemahkan menjadi 1000000000000101b. Semua koefisien, misalnya x2 atau x15 direpresentasikan sebagai logika 1 dalam nilai *binary*.

4. IMPLEMENTASI SOFTWARE CRC - 16 [1]

Ada dua teknik yang berbeda untuk mengimplementasikan sebuah CRC dalam *software*. Satu adalah implementasi *loop – driven* dan yang satunya lagi adalah sebuah implementasi *tabel – driven*. Yuanhong Huo et. al [11] menjelaskan pada publikasinya tentang perhitungan CRC parallel ber arsitektur *table – based* (*table – driven*).

Implementasi *loop – driven* bekerja seperti perhitungan yang ditunjukkan di gambar 8.



Gambar 8. Implementasi software CRC Loop - Driven

Data diumpankan ke sebuah *shift register*. Jika keluar sebuah binary 1 pada *most significant bit*, data tersebut di XOR kan dengan *generator polynomial*.

Metode yang lain untuk menghitung sebuah CRC adalah menggunakan nilai *pre calculated* dan meng XOR kan nya ke *shift register*.

IMPLEMENTASI CRC LOOP – DRIVEN

Bagian ini menjelaskan implementasi *software* CRC secara *loop driven*.

CRC Generation

Implementasi dari sebuah *loop driven* CRC *generation* ditunjukkan di gambar 8. Pada langkah pertama register register, CRC_HIGH dan CRC_LOW, diinisialisasi dengan dua byte pertama dari data. CRC_BUFF dimuati dengan byte data yang ketiga. Setelah itu, MSB dari CRC_BUFF digeser ke dalam LSB dari CRC_LOW dan MSB dari CRC_LOW digeser ke dalam LSB dari CRC_HIGH. MSB dari CRC_HIGH, yang sekarang disimpan di *Carry flag*, dites untuk melihat apakah dia di set. Jika bit tersebut diset, register register, CRC_HIGH dan CRC_LOW, akan di XOR kan dengan *generator polynomial* CRC – 16. Jika bit tersebut tidak diset, bit berikutnya dari CRC_BUFF akan digeser ke dalam LSB dari CRC_LOW. Proses ini diulang sampai semua data dari CRC_BUFF digeser ke dalam CRC_LOW. Setelah ini, CRC_BUFF dimuati dengan byte data yang berikutnya. Kemudian semua byte data diproses, dan 16 nol ditambahkan ke *message*. Register register, CRC_HIGH dan CRC_LOW, berisi nilai CRC yang dihitung. Panjang *message* tidak ditentukan.

CRC Checking

Pengecekan CRC menggunakan teknik yang sama seperti CRC *generation*, satu satunya perbedaannya adalah nol tidak ditambahkan ke *message*.

IMPLEMENTASI CRC TABLE-DRIVEN

Sebuah rutin CRC *table – driven* menggunakan sebuah teknik yang berbeda dari sebuah rutin CRC *loop – driven*. Ide di balik sebuah implementasi CRC *tabel – driven* adalah, sebagai ganti menghitung CRC bit demi bit, byte byte yang sudah dihitung sebelumnya di XOR kan ke data.

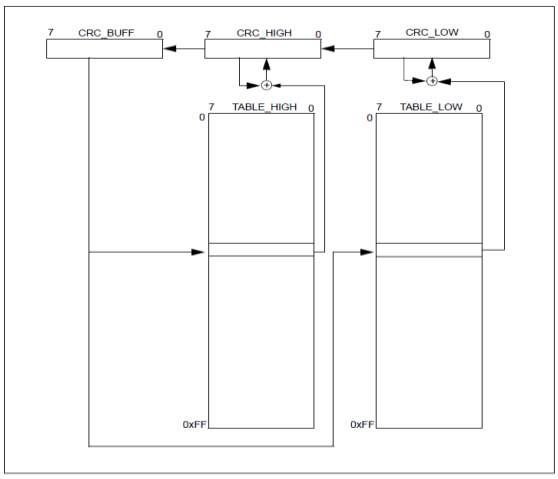
Keuntungan dari implementasi *table-driven* adalah bahwa dia lebih cepat daripada solusi *loop-driven*. Kekurangannya adalah bahwa dia mengkonsumsi memori program lebih banyak yang disebabkan oleh ukuran *look – up table*.

CRC Generation

CRC pada implementasi *table driven* dihasilkan dengan membaca sebuah nilai *precomputed* dari sebuah tabel dan meng XOR hasilnya dengan *low byte* dan *high byte* dari *shift register* CRC.

Pada langkah pertama, register – register, CRC_BUFF, CRC_HIGH, dan CRC_LOW, di inisialisasi dengan tiga byte data pertama. Setelah itu, nilai di dalam CRC_BUFF digunakan sebagai sebuah *offset* untuk mendapatkan nilai untuk nilai CRC *precomputed*

dari *look – up tabel*. Karena CRC – 16 panjangnya adalah 16 bit, *look – up tabel* dibagi ke dalam dua buah *look – up tabel* yang terpisah. Satu untuk *high byte* dari register CRC dan satunya lagi untuk *low byte* dari register CRC (lihat Gambar 9).



Gambar 9. Implementasi software perhitungan CRC - 16 Table - Driven

Hasil dari *look – up table* dari *high byte* di XOR kan dengan isi dari register CRC_HIGH. Hasil untuk *low byte* di XOR kan dengan isi dari CRC_LOW. Hasilnya disimpan kembali di CRC_LOW.

Langkah berikutnya adalah bahwa isi dari CRC_HIGH digeser ke dalam CRC_BUFF dan isi dari CRC_LOW digeser ke dalam register CRC_HIGH. Kemudian, register CRC_LOW dimuati dengan byte data yang baru.

Proses ini berulang untuk semua byte data. Pada akhir CRC *generation*, *message* harus ditambahi dengan 16 buah nol. Ke enambelas nol tersebut diperlakukan seperti byte data.

Setelah perhitungan selesai, isi dari register, CRC_HIGH dan CRC_LOW ditambahkan ke *message*.

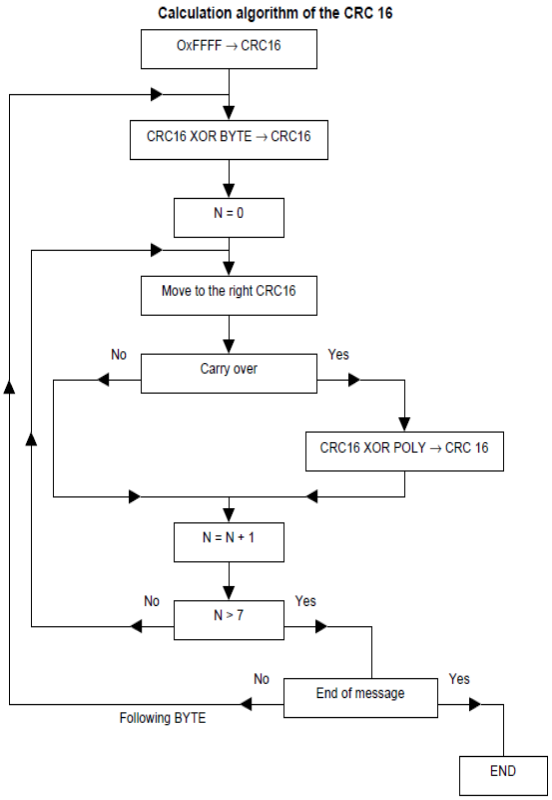
CRC Checking

CRC check menggunakan teknik yang sama seperti CRC *generation*, perbedaannya adalah tidak ditambahkan nol ke *message*.

5. HASIL DAN PEMBAHASAN

5.1. Rutin CRC – 16

Gambar 10 berikut menunjukkan sebuah algoritma umum untuk perhitungan CRC – 16.



Gambar 10. Algoritma perhitungan untuk CRC - 16

Keterangan gambar 10 :
XOR = Exclusive OR
N = Jumlah bit informasi
POLY = Perhitungan polynomial dari
CRC 16 = 1010 0000 0000 0001
(Generating polynomial = 1 + x2 + x15 + x16)
Pada CRC 16, byte pertama yang ditransmisikan adalah *low – order byte*

Di sini, dibuat sebuah implementasi *software* CRC-16 berbasis tabel (*table – driven*). Secara garis besar, algoritma perhitungan untuk CRC – 16 ditunjukkan di Gambar 10.

Dan Gambar 11 berikut ini adalah contoh rutin CRC-16 berbasis tabel yang dihasilkan :

```
// Fungsi CRC16 berbasis tabel
unsigned  char  calculate_CRC_Lo(unsigned  char
*message, int length, unsigned char *CRC)
{
 unsigned char CRCHi, CRCLo, TempHi, TempLo;

static const unsigned char table[512] = {
0x00, 0x00, 0xC0, 0xC1, 0xC1, 0x81, 0x01, 0x40,
0xC3, 0x01, 0x03, 0xC0, 0x02, 0x80, 0xC2, 0x41,
0xC6, 0x01, 0x06, 0xC0, 0x07, 0x80, 0xC7, 0x41,
0x05, 0x00, 0xC5, 0xC1, 0xC4, 0x81, 0x04, 0x40,
0xCC, 0x01, 0x0C, 0xC0, 0x0D, 0x80, 0xCD, 0x41,
```

```
0x0F, 0x00, 0xCF, 0xC1, 0xCE, 0x81, 0x0E, 0x40,
0x0A, 0x00, 0xCA, 0xC1, 0xCB, 0x81, 0x0B, 0x40,
0xC9, 0x01, 0x09, 0xC0, 0x08, 0x80, 0xC8, 0x41,
0xD8, 0x01, 0x18, 0xC0, 0x19, 0x80, 0xD9, 0x41,
0x1B, 0x00, 0xDB, 0xC1, 0xDA, 0x81, 0x1A, 0x40,
0x1E, 0x00, 0xDE, 0xC1, 0xDF, 0x81, 0x1F, 0x40,
0xDD, 0x01, 0x1D, 0xC0, 0x1C, 0x80, 0xDC, 0x41,
0x14, 0x00, 0xD4, 0xC1, 0xD5, 0x81, 0x15, 0x40,
0xD7, 0x01, 0x17, 0xC0, 0x16, 0x80, 0xD6, 0x41,
0xD2, 0x01, 0x12, 0xC0, 0x13, 0x80, 0xD3, 0x41,
0x11, 0x00, 0xD1, 0xC1, 0xD0, 0x81, 0x10, 0x40,
0xF0, 0x01, 0x30, 0xC0, 0x31, 0x80, 0xF1, 0x41,
0x33, 0x00, 0xF3, 0xC1, 0xF2, 0x81, 0x32, 0x40,
0x36, 0x00, 0xF6, 0xC1, 0xF7, 0x81, 0x37, 0x40,
0xF5, 0x01, 0x35, 0xC0, 0x34, 0x80, 0xF4, 0x41,
0x3C, 0x00, 0xFC, 0xC1, 0xFD, 0x81, 0x3D, 0x40,
0xFF, 0x01, 0x3F, 0xC0, 0x3E, 0x80, 0xFE, 0x41,
0xFA, 0x01, 0x3A, 0xC0, 0x3B, 0x80, 0xFB, 0x41,
0x39, 0x00, 0xF9, 0xC1, 0xF8, 0x81, 0x38, 0x40,
0x28, 0x00, 0xE8, 0xC1, 0xE9, 0x81, 0x29, 0x40,
0xEB, 0x01, 0x2B, 0xC0, 0x2A, 0x80, 0xEA, 0x41,
0xEE, 0x01, 0x2E, 0xC0, 0x2F, 0x80, 0xEF, 0x41,
0x2D, 0x00, 0xED, 0xC1, 0xEC, 0x81, 0x2C, 0x40,
0xE4, 0x01, 0x24, 0xC0, 0x25, 0x80, 0xE5, 0x41,
0x27, 0x00, 0xE7, 0xC1, 0xE6, 0x81, 0x26, 0x40,
0x22, 0x00, 0xE2, 0xC1, 0xE3, 0x81, 0x23, 0x40,
0xE1, 0x01, 0x21, 0xC0, 0x20, 0x80, 0xE0, 0x41,
0xA0, 0x01, 0x60, 0xC0, 0x61, 0x80, 0xA1, 0x41,
0x63, 0x00, 0xA3, 0xC1, 0xA2, 0x81, 0x62, 0x40,
0x66, 0x00, 0xA6, 0xC1, 0xA7, 0x81, 0x67, 0x40,
0xA5, 0x01, 0x65, 0xC0, 0x64, 0x80, 0xA4, 0x41,
0x6C, 0x00, 0xAC, 0xC1, 0xAD, 0x81, 0x6D, 0x40,
0xAF, 0x01, 0x6F, 0xC0, 0x6E, 0x80, 0xAE, 0x41,
0xAA, 0x01, 0x6A, 0xC0, 0x6B, 0x80, 0xAB, 0x41,
0x69, 0x00, 0xA9, 0xC1, 0xA8, 0x81, 0x68, 0x40,
0x78, 0x00, 0xB8, 0xC1, 0xB9, 0x81, 0x79, 0x40,
0xBB, 0x01, 0x7B, 0xC0, 0x7A, 0x80, 0xBA, 0x41,
0xBE, 0x01, 0x7E, 0xC0, 0x7F, 0x80, 0xBF, 0x41,
0x7D, 0x00, 0xBD, 0xC1, 0xBC, 0x81, 0x7C, 0x40,
0xB4, 0x01, 0x74, 0xC0, 0x75, 0x80, 0xB5, 0x41,
0x77, 0x00, 0xB7, 0xC1, 0xB6, 0x81, 0x76, 0x40,
0x72, 0x00, 0xB2, 0xC1, 0xB3, 0x81, 0x73, 0x40,
0xB1, 0x01, 0x71, 0xC0, 0x70, 0x80, 0xB0, 0x41,
0x50, 0x00, 0x90, 0xC1, 0x91, 0x81, 0x51, 0x40,
0x93, 0x01, 0x53, 0xC0, 0x52, 0x80, 0x92, 0x41,
0x96, 0x01, 0x56, 0xC0, 0x57, 0x80, 0x97, 0x41,
0x55, 0x00, 0x95, 0xC1, 0x94, 0x81, 0x54, 0x40,
0x9C, 0x01, 0x5C, 0xC0, 0x5D, 0x80, 0x9D, 0x41,
0x5F, 0x00, 0x9F, 0xC1, 0x9E, 0x81, 0x5E, 0x40,
0x5A, 0x00, 0x9A, 0xC1, 0x9B, 0x81, 0x5B, 0x40,
0x99, 0x01, 0x59, 0xC0, 0x58, 0x80, 0x98, 0x41,
0x88, 0x01, 0x48, 0xC0, 0x49, 0x80, 0x89, 0x41,
0x4B, 0x00, 0x8B, 0xC1, 0x8A, 0x81, 0x4A, 0x40,
0x4E, 0x00, 0x8E, 0xC1, 0x8F, 0x81, 0x4F, 0x40,
0x8D, 0x01, 0x4D, 0xC0, 0x4C, 0x80, 0x8C, 0x41,
0x44, 0x00, 0x84, 0xC1, 0x85, 0x81, 0x45, 0x40,
0x87, 0x01, 0x47, 0xC0, 0x46, 0x80, 0x86, 0x41,
0x82, 0x01, 0x42, 0xC0, 0x43, 0x80, 0x83, 0x41,
0x41, 0x00, 0x81, 0xC1, 0x80, 0x81, 0x40, 0x40,
};
  CRCHi = 0xff;
  CRCLo = 0xff;
  while(length)
  {
   TempHi = CRCHi;
   TempLo = CRCLo;

   CRCHi=table[2 * (*message ^ TempLo)];
   CRCLo=TempHi^table[(2*(*message^TempLo))+1];

   message++;
   length--;
  };

  CRC [0] = CRCLo;
  CRC [1] = CRCHi;
  return;
}
```

Gambar 11. Contoh rutin CRC – 16 yang dihasilkan

Pada rutin CRC – 16 di atas, nilai nilai *pre computed* untuk *low order byte* dan *high order byte* dituliskan dalam satu tabel.

5.2. Pengetesan CRC – 16 dengan pesan MODBUS

Seperti dijelaskan pada bagian pendahuluan, komunikasi antara transmitter dengan receiver bisa berupa transfer dalam bentuk text. Pada protokol MODBUS *Remote Terminal Unit*, terjadi pengiriman text yaitu berupa pengiriman *message* dari *transmiter* ke *receiver*. Text tersebut berupa kumpulan byte 8-bit. Tiap tiap byte 8-bit byte dalam pesan tersebut terdiri dari dua buah karakter (huruf) 4 - bit hexadesimal (hexadesimal artinya bernilai 0 sampai F).

Pesan MODBUS (*query message* dan *respon message*) disimpan dalam sebuah array dengan jumlah elemen array menyesuaikan dengan jumlah byte data yang ditransmisikan.

Tabel 2. Byte CRC_LOW dan byte CRC_HIGH yang dihitung oleh transmitter

Messages	CRCLow	CRCHigh
0x41, '-', 'P', 'a', 'n', 'a', 's', '-', 0x43, '-', 0x52, '-'	0x86	0xD6
0x41, '-', 'P', 'a', 'n', 'a', 's', '-', 0x43, '-', 0x51, '-'	0x86	0x26
0x41, '-', 'D', 'i', 'n', 'g', 'i', 'n', '-'	0x32	0xC3
0x42, '-', 'P', 'a', 'n', 'a', 's', '-', 0x43, '-', 0x52, '-'	0x82	0xD2
0x42, '-', 'P', 'a', 'n', 'a', 's', '-', 0x43, '-', 0x51, '-'	0x82	0x22
0x42, '-', 'P', 'a', 'n', 'a', 's', '-', 0x46, '-', 0x52, '-'	0x82	0x1E
0x42, '-', 'P', 'a', 'n', 'a', 's', '-', 0x46, '-', 0x51, '-'	0x82	0xEE
0x42, '-', 'D', 'i', 'n', 'g', 'i', 'n', '-'	0x26	0x33

Tabel 2 menunjukkan beberapa pesan MODBUS *Remote Terminal Unit*. Satu contoh *message* akan dibahas di sini, yaitu *message* pertama. *Message* pertama tersebut tersimpan

dalam sebuah array dimensi satu dengan nama tx_buffer_panas dengan jumlah elemen array sebesar variabel TX_BUFFER_SIZE . Lengkapnya adalah seperti berikut :

```
unsigned char tx_buffer_panas
[TX_BUFFER_SIZE] = {0x41, '-', 'P',
'a', 'n', 'a', 's', '-', 0x43, '-', 0x52,
'-' };
```

Message tersebut digunakan sebagai sebuah pesan oleh *transmitter*, yang pada protokol MODBUS berupa *master*, yang ditujukan kepada *receiver*, yang pada protokol MODBUS berupa *slave* tertentu. *Message* tersebut memberi *command* kepada *receiver* untuk meng ON kan sebuah aplikasi yang dikendalikan oleh nya sehubungan dengan keadaan `'P', 'a', 'n', 'a', 's',` . Bisa terlihat, message tersebut terdiri dari 12 byte.

Transmitter, yang bentuk *hardware* nya bisa berupa *microcontroller based development board*, akan mengirimkan pesan tersebut kepada *receiver*, yang bentuk *hardware* nya juga berupa *microcontroller based development board*. Pesan ditransmit via pin *transmitter* (TX) pada fasilitas USART *microcontroller unit*. Sebelum transmisi dilakukan, *software* akan menghitung terlebih dahulu *Frame Check Sequence* atau *Cyclic Redundancy Check* dari ke 12 byte di atas. Terdapat sebuah bagian pada *software master* yang akan memanggil (memanfaatkan) rutin CRC-16 contoh di atas. Seperti yang ditunjukkan pada potongan baris program berikut ini :

```
tx_buffer_panas[12]=
calculate_CRC_Lo(tx_buffer_panas, 12,
0x00);
```

Potongan baris program di atas menghitung *low order byte* CRC_LOW dari *message* 12 byte di atas. Hal yang sama dilakukan pada perhitungan *high order byte* CRC_HIGH.

Kedua variabel CRC, CRC_LOW dan CRC_HIGH selanjutnya ditambahkan ke message 12 byte di atas, dijadikan sebagai dua elemen terakhir dari array yaitu `tx_buffer_panas[12]` dan `tx_buffer_panas[13]`. Ketika CRC 16 – bit tersebut ditransmisikan, *low – order byte* akan ditransmisikan terlebih dahulu, diikuti dengan *high – order byte*.

Pada sisi receiver, *message* yang terdiri dari 12 byte data informasi dan 2 byte terakhir CRC akan diterima via pin *receiver* (TX) pada fasilitas USART *microcontroller unit*. *Receiver* selanjutnya secara *software* akan meghitung kembali byte CRC_LOW dan byte CRC_HIGH dari 12 byte data informasi yang telah diterima. Lihat tabel 3 berikut.

Tabel 3. Byte CRC_LOW dan byte CRC_HIGH yang dihitung kembali oleh receiver

Messages yang diterima	CRCLow	CRCHigh
0x41, '-', 'P', 'a', 'n', 'a', 's', '-', 0x43, '-', 0x52, '-'	0x86	0xD6
0x41, '-', 'P', 'a', 'n', 'a', 's', '-', 0x43, '-', 0x51, '-'	0x86	0x26
0x41, '-', 'D', 'i', 'n', 'g', 'i', 'n', '-'	0x32	0xC3
0x42, '-', 'P', 'a', 'n', 'a', 's', '-', 0x43, '-', 0x52, '-'	0x82	0xD2
0x42, '-', 'P', 'a', 'n', 'a', 's', '-', 0x43, '-', 0x51, '-'	0x82	0x22
0x42, '-', 'P', 'a', 'n', 'a', 's', '-', 0x46, '-', 0x52, '-'	0x82	0x1E
0x42, '-', 'P', 'a', 'n', 'a', 's', '-', 0x46, '-', 0x51, '-'	0x82	0xEE
0x42, '-', 'D', 'i', 'n', 'g', 'i', 'n', '-'	0x26	0x33

Apabila tidak ada *error* pada saat transmisi, nilai nilai CRC_LOW dan CRC_HIGH yang dihitung oleh *receiver* akan **sama** dengan nilai nilai byte CRC_LOW dan CRC_HIGH yang telah diterima. Apabila *error* terjadi selama transmisi data, nilai nilai CRC_LOW dan CRC_HIGH yang dihitung oleh *receiver* akan **berbeda** dari nilai nilai byte CRC_LOW dan CRC_HIGH yang telah diterima. *Receiver* selanjutnya akan meminta *re-transmisi* data. Jadi, skema *error detection* (yang kemudian bisa diikuti dengan meminta *re - transmission* data dari *transmitter*) terjadi di sini.

6. KESIMPULAN

Dari pembahasan di atas, dapat ditarik kesimpulan sebagai berikut.

Transmisi data bisa menimbulkan *error* pada byte data yang dikirim disebabkan oleh adanya *noise* pada kanal. *Error* bisa berupa *single bit – error*, *double bit - error* (dua *single – bit error* terisolasi satu sama lain), *odd number error*, atau *burst error*.

Sebuah *generator polynomial* yang didesain dengan baik bisa mendeteksi jenis - jenis *error* tersebut. CRC – 16 merupakan salah satu *generator polynomial* yang bisa diaplikasikan untuk MODBUS RTU. Implementasi *software* CRC -16 bisa dilakukan secara *loop – driven* atau *table – driven*. Pada paper ini, dibahas implementasi *software* CRC – 16 secara *table - driven*. Rutin untuk CRC – 16 bisa manjadi bagian dari program utama protokol MODBUS.

DAFTAR PUSTAKA

[1] AN730, "CRC Generating and Checking", Microchip Technology Inc., 2000

[2] Behrouz A. Forouzan, "Data Communications and Networking", Mc Graw Hill, 2007, pp 267 - 306

[3] Fred Halsall, "Data Comunications, Computer Networks, and Open Systems", Addison Wesley, 1995

[4] Hazarathaiah Malepati, "Digital Media Processing – DSP Algorithm Using C", NEWNES, 2010, pp 87 – 154

[5] H. Michael Ji, and Earl Killian, "Fast Parallel CRC Algorithm and Implementation on a Configurable Processor", Tensilica, Inc. 3255-6 Scott Blvd Santa Clara, CA 95054

[6] Jonathan Stone et. al, "Performance of Checksums and CRC's over Real Data", IEEE/ACM TRANSACTIONS ON NETWORKING, VOL. 6, NO. 5, OCTOBER 1998.

[7] Rofan Aziz, Karsid, "Uji Performansi Kontrol Suhu dan Kelembaban Menggunakan Variasi Kontrol Digital dan Kontrol Scheduling untuk Pengawetan Buah dan Sayuran", Jurusan Teknik Pendingin dan Tata Udara, Politeknik Negeri Indramayu dalam Jurnal JNTE Vol: 4, No. 2, September 2015

[8] Sunil Shukla, Neil W. Bergmann, "Single bit error correction implementation in CRC-16 on FPGA", School of ITEE, The University of Queensland, Australia

[9] Wael M El-Medany, "FPGA Implementation of CRC with Error Correction", Computer Engineering Department, College of Information Technology, University Of Bahrain, 32038 Bahrain, at ICWMC 2012 : The Eighth International Conference on Wireless and Mobile Communications

[10] Yoshihisa Desaki et.al, "Double and Triple Error Detecting Capability of Internet Checksum and Estimation of Probability of Undetectable Error", Graduate School of Engineering, Tokyo Metropolitan University Hachioji, Tokyo 192-03, Japan

[11] Yuanhong Huo et. al, "High Performance Table-Based Architecture for Parallel CRC Calculation", School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China

Biodata Penulis

Arief Wisnu Wardhana, lahir di Solo, Indonesia pada tanggal 30 Desember 1972. Menyelesaikan pendidikan S1 Engineering dalam bidang Electronic & Information dari University of Huddersfield, Huddersfield, The United Kingdom pada tahun 1997. Saat ini, penulis masih menjadi mahasiswa Pasca Sarjana S2 Jurusan Teknik Elektro Fakultas Teknik Universitas Gadjah Mada Yogyakarta. Penulis adalah dosen Jurusan Teknik Elektro, Fakultas Teknik Universitas Jenderal Soedirman, Purwokerto, Indonesia, bidang keahlian Sistem Isyarat dan Kendali. Interest utama pada area embedded system dan embedded programming.