INTRODUCTION

In most applications of serial links, because there is a significant chance of message corruption by electrical noise, some form of error detection and recovery system is required. Usually, error detection is effected by including a certain amount of redundant information with the data in each message, and recovery is by retransmission.

The choice of technique depends on factors such as:

- The level of security required.
- The nature of the data, such as the message length.
- Transmission speed.
- Transmission distance.
- The nature of the noise.
- Hardware and software facilities available.

CYCLIC REDUNDANCY CHECKS (CRC's)

The technique used in GEM 80 standard serial links transmission is to use a cyclic redundancy check (CRC) of 16 bits. To achieve error detection,this requires that an extra 16 bits (2 bytes) are included with each message transmitted.

How a CRC works

A CRC code is similar to the remainder you get if you do a division sum. To get the idea, consider doing such a sum, not in binary as used in a GEM 80 Controller or Programmer or in a computer, but in decimal. Suppose, for instance, that you wish to send a message by Telex where it is vital that a number is tranmitted correctly. You therefore need a confidence check to make sure that the Telex operator has keyed the number correctly and that the number has not been distorted during its transmission over the public telecom network. What we need, at both the transmitting and reciving ends, is a common key number. Suppose that, as an example, it is 97.

Now suppose that we want to transmit the number 144752. If we divide this by 97, we get 1492 with a remainder of 28. We therefore transmit the original number 144752 with the remainder added to the end as a couple of check digits, i.e. we transmit 14475228.

Note... If the remainder were less than 10, we would still make it up to two digits, e.g. we would transmit a remainder of 7 as the two digits 07.

At the receiving end, we know that the last two digits of the number 14475228 when received are check digits, so we strip them off and get the value 144752. We also know that the key number is 97, we now divide 144752 by 97 and check that we get a remainder equal to 28. Since we do indeed get the right answer, we are fairly confident that the number has been transmitted without any error.

Suppose, however, that the 7 had become changed into an 8, and that the number received was 14485228. Dividing 144852 by 97 gives us a remainder of 31, which does not tally with the two check digits of 28, and we know that there has been a transmission error.

The only case where the simple decimal check we have used above will fail is where the number becomes distorted in such a way that it equals the original number plus or minus a multiple of 97. For instance, if 144752 had become altered to 145722 (144752 + 970), our check would fail to detect the error, as we still get the remainder of 28 when we divide by 97.

Note... To get such an error with the key of 97, we must have a minimum of 2 digits in the number in error and in most cases 3 digits in error.

With a CRC, we do something similar to the decimal check given above except that, first, we are working in binary and not in decimal, and second, we perform a type of division sum with no carries, known technically as modulo 2 division.

Choice of Generator Polynomial

The key binary number we use in GEM 80 systems, equivalent to the 97 used in our decimal example, is 11000000000000101, which is a 17-bit number, and makes sure that we got a 16-bit remainder which can be transmitted as 2 bytes. The divisor is referred to technically as the 'generator polynomial', and is usually written in articles and books on the subject of error checking codes, not in powers of 2, but in powers of X, i.e. as:

$$X^{16} + X^{15} + X^2 + 1$$

This, of course, is not the only possible polynomial that could be used, just as 97 is not the only possible number we could have used in our decimal remainder check example. Any 17-bit binary number, giving us a 16-bit remainder, would provide us with a 2- byte check code. However, some polynomials are better than others for detecting particular types of error, so that there is something of an art in deciding what are the most likely types of message corruption which will occur.

Returning to the previous decimal example, where we found the remainder resulting from dividing the number to be transmitted by a key number, we arbitrarily chose 97 as the key number. Was this a good choice?

Let us consider some of the possible types of error we could get. First, we note that the remainder from dividing by 97 will be a 2 digit number in the range 0 to 96. If errors were to occur purely at random, there would be a 1 in 96 chance of the remainder checking out when the number was corrupted. However, it is likely that most of the message corruptions will be keying errors. The Telex machine operator might hit an adjacent key that was 1 too big or too small, or might transpose two digits.

Hitting a key that is out by one will give a number that is 1, 10, 100, 1000, etc., too large or too small. An extremely bad choice of key number would therefore be 50, say, since the remainder resulting from dividing by 50 would not show up any out-by-one errors except in the last two digits.

Further, once having hit a key which is one to the right or left of the correct one, the operator might send two or three digits in succession 1 too large or too small. For instance, if one of the operator's hands drifted out of position to the right, making digits 1 too big, then numbers might be out by 11, 110, 1100, 11000 etc., or maybe by 111, 1110, 11100 etc. The first case would make 55 a bad choice, while the second case would make 37 or 74 a bad choice.

In a similar way, with GEM 80 serial links, the designers first considered the most likely forms of corruption to the messages passing along them. Corruptions are most likely to be produced by electrical interference from contactors producing arcs as their contacts break inductive circuits. Such arcing is likely to produce a burst of errors, rather than random single bit errors.

From the possible 16-bit CRC checks which could have been used, the particular one chosen for GEM 80 (which is also used on IBM's BSC protocol) will detect the following errors:-

- All single-bit errors.
- Any odd number of errors.
- All single and double errors as long as the message length is no longer than 32,767 bits (which is just over 4,000 bytes).

Note... Message lengths in GEM 80 systems rarely exceed 128 words, or 256 bytes.

- Any two burst errors of two bits as long as the message length is no longer than 32,767 bits.
- Any single burst of length 16 bits or less.
- All but 1 in 32,768 single bursts of length 17 bits.
- All but 1 in 65,536 single bursts of length greater than 17 bits.

CALCULATION OF CRCs

Users often want to couple a computer or other intelligent equipment to GEM 80 controller serial links. With standard GEM 80 serial ports emanating from processor modules, the message protocol is fixed, and will either be ESP or CORONET, as described in the previous Items in this Section. Each message ends with two CRC characters. The polynomial used is the CRC16 polynomial given previously.

Where you need to communicate using a different protocol, you need a programmable serial communications module in your GEM 80 controller. You can program this module using the COMMUNE language for more-or-less any protocol you need, although there are some data rate limitations. The COMMUNE language, a subset of BASIC, includes built-in functions for evaluating the CRC16 polynomial given previously, and also the CCITT polynomial:

$$X^{16} + X^{12} + X^5 + 1$$

A third option available in COMMUNE is longitudinal parity checking. This is, in fact, equivalent to using the polynomial:

$$X^8 + 1$$

If you require any other form of error detection, you can program it using COMMUNE language instructions. However, the execution of your instructions will, take longer than the built-in functions.

IMPLEMENTATION OF CRC GENERATION

It is unlikely that you will want to use hardware to generate or check CRCs. The overhead imposed by using software is relatively minor. As long as the CRC bytes are sent as the last two characters of the message, the CRC calculation can proceed a byte or a word at a time. You do not therefore need to know the complete message, but you can perform the evaluation as a running calculation.

Algorithm used in GEM 80 Systems

The algorithm is described in:

'On the Computation of Cyclic Redundancy Checks by Program'
P. Higginson and P. Kirstein
University of London Institute of Computer Science
INDRA Note No. 240
March 1972

and is as follows:-

Let: M_n = nth message byte

H_n = high residue byte before nth message byte is processed

L_n = low residue byte before nth message byte is processed

H_{n-1} = high residue byte after nth message byte is processed

L_{n-1} = low residue byte after nth message byte is processed

Then:

$$H_{n-1}.x^3 + L_{n-1} = L_n.x^3 + (H_n + M_n).(x^2 + x) + P_n.(x^{15} + x + 1)$$

where:

P_n = parity of $(H_n + M_n)$

Notation:

All '+' symbols mean exclusive OR (not ADD).

Any byte multiplied by x^n means that the byte is shifted left by n bits.

'Parity' means a bit that is set to 1 if the quantity of 1's in the byte is an odd number, or is set to 0 if the quantity of 1's in the byte is an even number.

Note... 1. The left-hand side of the formula represents a 16- bit word, where the lower byte contains L_{n-1}, and the upper byte contains H_{n+1} since this byte has been shifted 8 bits left.

2. The second and third terms of the equation are both dependent solely on the byte $(H_n + M_n)$. Given sufficient memory, these terms can be stored in a look-up table of 256 16-bit words, or 512 bytes of memory. Although requiring more memory, this provides the fastest implementation of the algorithm. It also avoids problems in implementation where the particular microprocessor you are using has no jump-on-parity instructions in its instruction set.

Description of Implementation Examples

Three examples of sub-routines in assembly code are given at the end of this Item. The first example shows an implementation in 8086 code; the second shows an implementation in 8085 code; and the third shows an implementation in 8086 code using a look-up table. The 8086 code is the more straight-forward, as the 8086 is a 16-bit processor. Since the 8085 is only an 8-bit microprocessor, it performs the algorithm using a number of correction factors.

Note... When performing modulo 2 division by a generator polynomial, this division must be carried out on the *actual string of bits transmitted* down the serial link. Bits are not normally transmitted by USART chips in the same order as they are stored in a microprocessor memory or registers, but in the reverse order. Also, the highest power of the polynomial divisor respresents the most significant bit and will therefore appear at the left- hand side.

In the examples of implementations of the algorithm, you will therefore find that right shifts or rotates are used where you might at first sight think left shifts or rotates ought to have been used.

Also included after the examples are flow charts corresponding to the examples, which you may find more useful when implementing a CRC routine on processors of manufacture other than Intel.

MESSAGE EXAMPLES

The following examples of messages are provided so that you can check any implementation of the CRC algorithm for processors or computers other than the Intel range.

All the codes given below are in hexadecimal:-

Example 1:

02 03 4B 4A 51 42 11 32 29 18 15 43 71 1A 4C 3D 35 4D 3B 21 29 39 77 44 03

CRC = F29C

Notes...

In the above example, and in all the following examples, the initial 02 in hex represents STX, the starting character for any message using ESP. This STX character must *not* be included in the CRC calculation.

However, *all* other characters [including any DLE characters (10 hex)] *must* be included in the calculation.

The examples use the standard message structure for J/K table exchange given in Item 13. The characters after the STX begin with 03 (tributary 0, message type 3), followed by 4B,4A (ASCII K and J), followed by the data. The message examples end with 03 (ETX in hex), which in practice would alternate with ETB characters on successive messages via the same route. This ETX character would then be followed by the CRC we are calculating.

Finally, a very important point: The two bytes making up the CRC *must be* transmitted in reverse order, e.g. in Example 1 above, the CRC must be transmitted as 9C F2.

Example 2:

02 03 4B 4A 34 11 76 37 52 25 00 1B 1A 10 05 46 00 41 4D 03

CRC = 8CCF

Example 3:

02 03 4B 4A 66 3C 14 23 45 45 31 81 75 74 60 70 10 17 00 15 38 03

CRC = A0F5

Example 4:

02 03 4B 4A 08 A7 1B 06 00 7F 5F 6F 08 66 1C 1B 3F 6F 03

CRC = DA40

Example 5:

02 03 4B 4A 34 56 76 11 52 37 00 23 1A 1B 55 09 00 46 4D 41 28 29 45 2F 03

CRC = AA26