3 PORT INTERFACE
RS-485 to Ethernet Converter

Only $170

RS-485 to Ethernet Converter

Powerful feature

- Protocol converter RS485 between Ethernet
- Offer TCP/IP Communication to Devices with RS485 I/F

<table>
<thead>
<tr>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network: TCP, UDP, DHCP, ICMP, IPv4, ARP, IGMP, PPTP, Ethernet, Auto MDI/MDIX, 10/100 Base-TX Auto negotiation (Full/half Duplex)</td>
</tr>
<tr>
<td>Serial: RS485 3 Ports, 1,200~115,200 bps, Terminal block I/F Type</td>
</tr>
<tr>
<td>Control program: IP Address &amp; port setting, serial condition configuration, Data transmit Monitoring</td>
</tr>
<tr>
<td>Accessory: Power adapter 9V 1500mA, LAN cable</td>
</tr>
<tr>
<td>Etc: DIP Switch (485 Baud Rate setting), LED: Power, Network, 485 Port transmission signal</td>
</tr>
</tbody>
</table>

www.icbank.com
Error Checking

Data security isn’t a perk. It’s a requirement. This article covers the topics of error checking, checksums, and the cyclic redundancy check. Error checking matters.

Remember when we used highly condensed polycyclic aromatic hydrocarbon [bitumen] for its sealing properties? Oh, you weren’t around B.C.? Maybe you remember using sealing wax to protect your document. Oh, you no longer wear that signet ring? Do you have a PIN? Most people today have more than one. While tampering may be on the minds of many people, it is often the integrity of the data itself that is of most concern. As most data is exchanged via some sort of serial link, today we employ various means of examining its integrity. This might be a byte at a time in hardware, as between UARTs, or by the packet via software. A UART can automatically append an extra parity bit to each byte it transfers, as an indication that the data was received without error. A single bit error (or at least an odd number of bit errors) can produce an error flag.

When sending data within a network, the actual data is often broken into pieces. Each piece is wrapped by the open systems interconnection (OSI) model so that its correct reception can be ensured and the piece can be reassembled with others back into the original data. Each wrapper contains its own technique to ensure packet integrity. The most common approach is to use a checksum or a cyclic redundancy check (CRC).

An application will have little control over packets and how they employ packet integrity. However, the same techniques can be used by your application to protect its data. Checksums can be pretty simple to use. You can think of a UART using parity as the equivalent of a 1-bit checksum. Each of the 8 data bits, either a 1 or a 0, is added together:

$$\text{Byte} = 0600110001$$
$$0 + 0 + 1 + 0 + 0 + 0 + 1 + 3$$

This total AND 1 equals the parity of the byte:

$$3 \text{ AND} 0b1 = 0b1$$

An alternate approach to this is to XOR each bit value with the previous result:

$$\text{Byte} = 0600110001 \text{ (result initialized to 0)}$$
$$0 \text{ XOR result (0) = 0}$$
$$0 \text{ XOR result (0) = 0}$$
$$1 \text{ XOR result (0) = 1}$$
$$1 \text{ XOR result (0) = 0}$$
$$0 \text{ XOR result (0) = 0}$$
$$0 \text{ XOR result (0) = 0}$$
$$1 \text{ XOR result (0) = 1}$$

The second approach might not seem any simpler, but it has the advantage of not requiring a huge accumulator, which may be required when using the first approach, if the data string is large. Whereas the UART’s parity spans a single byte, strings or blocks of data often use byte, word, or larger checksum values. When using a byte checksum, the total of all the bytes in the string (or block) is ANDed with 0xFF [limited] to 8 bits wide. A word checksum can handle more bytes before being limited by its 16-bit width.

While any rollover of a checksum’s width decreases its effectiveness in indicating a data...
Table 1—Here are a number of popular 16-bit CRC forms

<table>
<thead>
<tr>
<th>Name</th>
<th>Polynomial</th>
<th>Used In</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC-16-IBM</td>
<td>( x^{16} + x^5 + x^2 + 1 )</td>
<td>Bisync, Modbus, USB, ANSI X.3.28, many others; also known as CRC-16 and CRC-16-ANSI</td>
</tr>
<tr>
<td>CRC-16-CCITT</td>
<td>( x^{16} + x^5 + x^1 + x^0 )</td>
<td>X.25, HDLC, XMODEM, Bluetooth, SD, many others; known as CRC-CCITT</td>
</tr>
<tr>
<td>CRC-16-DNP</td>
<td>( x^{16} + x^{13} + x^2 )</td>
<td>DNP, IEC 607, M-Bus</td>
</tr>
<tr>
<td>CRC-16-DECT</td>
<td>( x^{16} + x^{10} + x^7 + x^3 + 1 )</td>
<td>Cordless telephones</td>
</tr>
</tbody>
</table>

error, it still boils down to the odds. That's "odds" as in, "Are you a betting man?" and not odds and evens. Errors that develop between the transmission and reception are dependent on the transmission medium and the environment. The cause is not the focus of the discussion, as the design and adoption of these transmission systems give us confidence in their low inherent error rates. However, implementing a data integrity method can result in identifying an integrity issue. The simplicity of checksums makes it ideal for basic error checking. Some simple errors can be missed by the checksum method. The same issue with two errors in a parity-checked byte exists with a checksum. If errors occur in the same bits of two different bytes, a checksum error won't be found. For example: 240 + 33 = 273, which has the same checksum as: 241 + 32 = 273.

The checksum's width makes no difference [8 bits, 16 bits, or even 64 bits]. A checksum is incapable of recognizing that this kind of error has occurred.

The context of the data can often help point out errors. For example, it is fairly easy to see a bit error in a string of text. But, if the data is a binary block, there is no context from which you can extrapolate the correction information.

I've got a couple of projects coming up that use a CRC instead of a checksum as a method of error detection. This seems to be a good time to look at how this works.

**CYCLIC REDUNDANCY CHECK**

A CRC is a more effective means of error detection than the checksum method because each piece of data (nominal a byte) can affect the full data width of the CRC, rather than just its own data width. CRCs can determine a single-bit error, a two-bit error, any odd number bit error, and many burst errors. To accomplish this, the additive method of calculating checksums is replaced with a division method. If we consider the data as the dividend, what determines what the divisor is?

A divisor is dependent on the width of the CRC and the polynomial chosen for its effectiveness based on probable error types. I won't discuss this further here. (If you're a math enthusiast, refer to the titles in the Resources section at the end of this article.) There are multiple standard divisors for 16-bit CRCs. Because we are dealing with base-2 or binary math, the divisors are given as polynomials. The CRC-16 standard uses \( 2^{16} + 2^{15} + 2^{2} + 2^{0} \) or 0b11000000000000101 (0x18005). The X.25 standard uses \( 2^{16} + 2^{15} + 2^{5} + 2^{3} \) or 0b100010000000000001001 (0x1021). You may have noted that each of these is actually 17 bits in width, but you will see shortly that the MSB (and it is always a "1") can be discarded, allowing the remaining 16 bits to be held in a word register. For the remainder of this discussion, I'll use the CRC-16 standard as the (polynomial) divisor.

Note that the divisor (polynomial) is 17 bits long. The MSBit is in red. The MSBit of the dividend determines whether we subtract the polynomial (if it's a "1") or 0. This choice always produces an MSBit of "0" in the result (which reduces the result's width). Bring down the next digit (in green) and the result again gains a bit of width. The subtraction (normally used in long division) is simplified to an XOR function (disregarding carries). After repeating the function eight times (once for each bit of original data), the final result (remainder) is the CRC value for the data. The quotient is not used.

So what about that 17th bit? Look back at the previous example. If we start by shifting the data (dividend) left into a 16-bit register 17 times, we end up with the first bit shifting out (into the carry) and the 2nd through 17th bits (in blue) in the 16-bit register. If the carry holds a "1," then the [16-bit] polynomial is XORed with the dividend register. If the carry holds a "0," then a [16-bit] zero is XORed with the dividend register. You'll note that you don't really have to XOR 0, as the result is equal to the
Table 2—This parameter model for two CRC-16 and two CRC-32 variants shows how drastically they can alter the resulting CRC. The last two parameters are optional, but they include a way of determining that the algorithm produces the desired effect. These include a string of test data (i.e., 0x31, 0x32, 0x33...) and the resultant 16/32-bit CRC.

dividend. At this point, the result is the CRC for one bit of data.

Should there be more data [bits], we prepare to repeat the function by shifting in the next bit of data, which forces out a new value into the carry. By this you can see that in the dividend the 17th bit is held in the carry. The 17th bit of the divisor [polynomial] isn’t really needed, it’s implied. When the carry is “1,” the 17th bit of the polynomial [which is always a “1”] will always be eliminated by the XOR. So, we don’t need it. This enables the actual polynomial 0x18005 to become 0x8005, a nice 16-bit value.

PARAMETERIZED MODEL

I previously mentioned CRC-16 and X25 as two 16-bit standards using different polynomials. In fact, there are more where these came from. While the polynomial itself is a good start (see Table 1) to providing a CRC routine for encoding and decoding your data, there are often other factors involved. A parameterized model can present these possibilities in a simple list form. This list, illustrated in Table 2, shows the polynomial discussed earlier, along with its potential variants. These include the initial value of the CRC, a value that must be XORd with the final calculated CRC, whether the data [input bytes] are to be initially reflected (LSBit first), and if the final CRC value must be reflected. Optional test data helps to give some level of confidence to your algorithm.

CALCULATING

The algorithm for encoding a CRC is the same for decoding it. The data is fed in and the algorithm produces a CRC. The calculated CRC is appended to the data at the

Figure 1—To calculate a CRC value for data, the algorithm must perform some process based on each bit of the data. The resultant CRC value remains a constant predetermined width.

Figure 2—Based on previously calculated CRCs for all byte values, a look-up table can speed up on-the-fly CRC calculations by reducing bit treatment to byte treatment.
transmission and stripped off at the reception end. If the received data is passed through the same algorithm, then the calculated CRC should match what was appended to the data.

Figure 1 depicts a possible algorithm for finding a 16-bit CRC value for a string of data. Each data byte in the string (or buffer) gets reflected, if necessary, and 16 bits of 0 are appended to the data. There are two loops within the algorithm: an outer loop to retrieve each data byte and an inner loop to shift each bit of the data byte into the WorkingWord register. The bit that pops out of the other end is used to determine whether or not to XOR the polynomial with the WorkingWord. This shifting and XORing is repeated until all the bits of all the bytes have been operated on. The result left in the WorkingWord register becomes the CRC, unless either of the two remaining operations changes it further.

Calculating a CRC for a long string of data can take some time. If you will be doing a lot of CRC work, or if it just needs to be done faster, you can consider other options. If you are willing to give up some program space, a CRC table might be the right approach. You will need to decide how much space you are willing to give up for table use. For a 16-bit CRC, tables will range from 16 words for a nibble table to 256 words for a byte table. Anything larger will cost just too much valuable space.

Figure 2 shows how the algorithm for a byte table might be used to speed up the CRC calculations. The inner loop in Figure 1 has disappeared and it is replaced by a precalculated table holding the results of all possible byte CRC calculations. A similar algorithm can be used based on nibbles. The nibble table is much smaller, but you will have to do twice as many lookups for the same amount of data.

**BUILD ME A TABLE**

I like building some quick sample code to make sure I understand the algorithms. Liberty Basic gives me a good playpen for fooling around with code. One benefit to this is the ability to easily create a table for any variation of a 16-bit CRC form. I started this BASIC application to simply try out the algorithm shown in Figure 1. While I started by hard coding in the constants of a specific 16-bit parameter model, once I got correct results with the algorithm, I had to make it more flexible. I added a drop-down menu so each parameter can be changed. The application uses the CRCTestString as default data and pumps it through the algorithm (looping through each bit of each byte of the string), finally comparing the result to the CRCTestCRC. Photo 1 shows a run displaying only the proof results. The proof can be displayed as well (see Photo 2), which shows operations on each bit, broken down into byte-sized chunks.
into any application in which you want to implement this method of calculating CRCs.

Note that the first 16 values are also the CRCs for a nibble table. Suppose your data is the single-byte 0xFF. After padding and reversing the data, you'd have 0xFF0000. WorkingWord is initialized with 0xFF00. The fifth nibble is pushed into the WorkingWord, and our pops the nibble 0xFF, leaving 0x0000 in the WorkingWord. The table entry at offset 0xF is 0x0022 [see Photo 3]. 0x0000 XOR 0x0022 = 0xF022. The sixth nibble is now pushed into the WorkingWord and out pops 0xF, leaving 0x0220. The table entry is again 0x0220. 0x0220 XOR 0x0022 = 0x0202. Why am I showing you this? Well, if you look at the last table entry in the byte table [see Photo 3] you'll see that 0x0202 is indeed the CRC for the byte 0xFF. This shows the relationship between nibble calculations and byte calculations. All of this is based on bit calculations for finding the CRC of a string of data.

APPLICATION DOWNLOAD

You can download a copy of this application from the Circuit Cellar FTP site and learn how CRCs can be created. This is a brute-force implementation, so please be kind. I know you “C” guys probably have canned routines for doing CRCs within your applications, and that’s fine. I present this for those of you who are interested in knowing more about the inner workings. Some of us are interested in what’s inside the black box.

WHO KNOWS?

Methods like the checksum and the CRC provide a level of confidence to data transfer integrity. You need to be confident that what goes into a transmitter is exactly what comes out of the receiver. Biologically, DNA goes through this process during reproduction. To replicate, DNA must unzip into two half strands [like a ladder with breaks through each rung]. Each half must be rebuilt as an exact copy of the original or you have an error, chromosome damage. What is the term used to describe the error-checking process that goes on to protect against DNA integrity errors? [8]

Jeff Bachiochi (pronounced BAH-key-AH-key) has been writing for Circuit Cellar since 1988. His background includes product design and manufacturing. You can reach him at jeff.bachiochi@imaginenow.com or at www.imaginenow.com.

PROJECT FILES


RESOURCES

A. S. Tanenbaum, *Computer Networks*, Prentice Hall, 1981. (See Section 3.5.3 on pages 128 to 132.)