

Parity, Checksums and CRC Checks

One of the issues which must be faced in any system is the problem of errors. We make an assumption - which is often justified - that a digital bit pattern remains constant in time and therefore information does not “decay away”. Error-checking is a device we use to confirm our prejudices or alert us to a failure in the system.

Virtually all forms of error checking involve adding something to the digital pattern. This increases the number of possible patterns (adding one bit to a pattern doubles the number of possibilities) If we then place rules on valid patterns we can arrange things so that valid patterns do not become other valid patterns through small errors.

Parity

Parity is the simplest form of error checking. It adds one bit to the pattern and then requires that the modulo-2 sum of all the bits of the pattern and the parity bit have a defined answer. The answer is 0 for even parity and 1 for odd parity. An alternative way of making the same statement is that odd(even) parity constrains there to be an odd(even) number of “1”s in the pattern plus parity bit.

Parity bits are sufficient to catch all single errors in the pattern plus parity bit as this will change a single 1 to a 0 or vice versa and therefore upset the parity calculation. A repeat of the parity calculation at any time will reveal this problem. However the system will not catch any double errors (except the trivial case of two errors on the same bit) and these will be flagged as valid.

For example a pattern of 0110100 becomes 00110100 with the addition of a parity bit and enforcement of odd parity. A single error would (for example) change the pattern to 00110000 which has the wrong parity. However a further error to 10110000 looks OK - but is in fact wrong.

The length of pattern which each parity bit “guards” should be as long as possible so that the parity bit takes up the least amount of extra storage, but not so long that a double error becomes likely. Thus if the probability of a single bit being is error is 10^{-6} , then the probability of an error in an 8-bit pattern (7 + parity) is about 8×10^{-6} and the probability of a double error is about 3×10^{-11} which is quite small.

An additional issue is that in some circumstances when an error does occur, it occurs

in a clump - ie groups of bits may get corrupted together. Thus if the probability of one bit being corrupted is P (small) then the probability of two successive bits being corrupted is not P^2 , but something much larger, maybe even approaching P . A way around that is to “matrix the check as follows

D11	D12	D13	D14	D15
D21	D22	D23	D24	D25
D31	D32	D33	D34	D35
D41	D42	D43	D44	D45
P1	P2	P3	P4	P5

In this matrix the parity bits P1-P5 are computed through the columns whereas the data is corrupted (for whatever reason) by rows. In this case a “burst error” of D22, D23 will produce parity errors on recalculation for bits P2 and P3 whereas a double error in row two would “fool” the system if parity were computed in rows.

(The astute reader will realise that by computing parity in both columns and rows, detection of errors in two dimensions is secured)

Checksums

Checksums are exactly the same as parity with two changes: The number of bits in the sums is larger and the result is always constrained to be zero. In other words to create the checksum of a pattern of entities of length n bits, an n -bit entity (the checksum) is added and the modulo 2^n sum of the entities and the checksum is constrained to be zero.

In this example the five bytes in the left-hand column are constrained to total zero - ie the last byte has been set to the negative of the modulo-256 column sum above it. The total shown in the last row is therefore 0. In the middle column, the second word has been corrupted by the changing of the two most-significant bits. This alters the sum and flags the error.

10110001	10110001	10110001
00011101	11011101	00011101
10000111	10000111	10000110
01111111	01111111	01111111
00101100	01101100	00101101
00000000	11000000	00000000

Unfortunately the checksum method can only reliably catch single-bit errors as there are still some errors which slip through - as you can see in the last column of the above example.

Cyclic Redundancy Checksums (CRCs)

CRCs seek to improve on checksums by increasing the complexity of the arithmetic. After all we increase the number of available patterns by 2^n by adding the checksum and there are only n single-bit errors per pattern and $n(n-1)/2$ double bit errors. If $n + n(n-1)/2 = n(n+1)/2 < 2^n$ then we can potentially detect all double-bit errors

The procedure for generating a CRC relies on a predetermined word of m -bits called the "generator" $P(x)$. The pattern itself is represented as $G(x)$. Both patterns are regarded as positive numbers. The steps are as follows...

- 1) multiply the $G(x)$ by 2^{m-1} (equivalent to a left-shift of $n-1$ places in binary) to form the message $M(x)$
- 2) divide $M(x)$ by the generator and take the remainder $C(x)$
- 3) Add the quantity $P(x)-C(x)$ to $M(x)$ giving $M'(x)$

The resulting $M'(x)$ is exactly divisible by $P(x)$ permitting a corresponding computation at a future time/place.

The trick is to produce the right $P(x)$ and much work (mathematical) has gone into producing such numbers. Here are a few...

CRC-12	110000000111
CRC-16	1100000000000010
CRC-CCITT	1000100000010000
CRC-32	is a bit more complex - but similar

A CRC check can catch all single, and a large number of other, errors. It is not prone to the “bursting” problem above. It is used extensively in disk systems, communication systems and other places where a check on a pattern has to be maintained.

Note that although I have described the arithmetic of the generation of CRCs in conventional terms, the actual calculations can be speeded up enormously by the use of fast algorithms or hardware solutions. However if you want to know about you will have to do some reading on your own.

Communications

I want to talk a bit about communications. This is an important subject in this field as many devices want to "talk" to other devices with hopefully as little confusion as possible. Communications can take many forms but we can classify these into two types: Point-to-point where only two devices are involved and "bus" where many devices are concerned.

I'm also going to make a distinction between the "rules" of the communication system and the physical medium which is used to implement these rules. All of these systems could be implemented with paper and pencil using Canada Post as the transmission medium as well as in electronic format. This means that the "rules" can be more durable than the technology and therefore do not change as often. In addition it means that information can be sent over a system which employs several physical links without elaborate transformation.

Point-to-Point Communications

Point-to-point communications may be subdivided into:

- non-checked
- error-checked and flag errors
- error-checked and re-transmit on error
- error-checked and corrected and flag errors
- error-checked and corrected and re-transmit on error

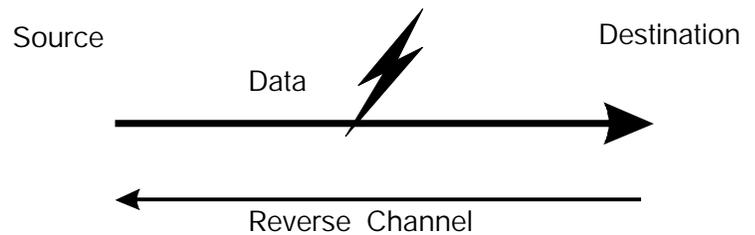
The simplest is obviously non-checked where one device sends to the other device and does not worry at all about whether the data reaches the other end. A TV transmitter is a classic example because it functions whether or not there are any TV sets receiving the signal. The advantage of this scheme is simplicity because there is no "reverse-channel" capability required. A further advantage is efficiency because no message time is wasted trying to recover from error situations. However there is no possibility of recovery from an error situation (and no possibility of discovering one because nobody is checking).

The next most advanced scheme is to check for errors and flag when one is detected.

This requires that the error can be detected using, as the simplest example, a parity check, or a CRC check for more elaborate schemes.

The message is checked on receipt and in the case of an error the message is rejected. A simple example of this is a standard computer terminal operating with parity on every character which can be set up to print a "?" character if there is an error in a character received. In redundant or non-important messages this provides a reasonably cheap and simple way to flag errors.

A more complicated scheme requires the use of a "reverse channel" by which the receiver can tell the transmitter that there has been an error and request a re-transmission. Thus the time scheme is as shown in the diagram. There obviously has to be some agreement about how many attempts can be made before giving up otherwise the system can "loop on error" for ever.



Simple Reverse Channel System

A yet more sophisticated scheme involves the embedding of enough information in the "message" that a reasonable error can not only be detected but also corrected. These schemes, of which the "Hamming code" is the most popular for simple cases, necessarily involve the addition of large amounts of error-checking information to the data stream and are incapable of correcting all errors - hence the options to flag uncorrectable errors and to retransmit on uncorrectable. A simple example of such a system is the "teletext" services which transmit data in TV pictures. These have to be heavily error detected/corrected as the error-rate of transmission is high and there is no possibility of a reverse channel so flagging uncorrectable errors is the only option available.

Hamming Codes

Hamming codes are interesting in that they extend the idea of parity to include an error correction as well as an error-detection scheme. The essential idea in a Hamming code system is that a unique number is generated by parity errors which uniquely identifies the bit which is in error. Since bits can only take two values, the correct value is then known.

Consider a 16-bit word with parity bits P1-P5 being generated as shown

		WORD BITS															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
P1		X		X		X	X		X		X		X	X		X	X
P2				X	X		X	X			X	X		X	X		X
Parity Bits P3		X	X				X	X	X	X				X	X	X	
P4							X	X	X	X	X	X	X				
P5		X	X	X	X	X											

[An "X" indicates that that particular bit of the word is included in the generation of the corresponding parity bit]

Each of the bits in the word is carried in at least two of the parity bits. If a single-bit error occurs then a unique set of parity errors occurs to tell you which bit is in error. But what happens if the parity bit is in error? In that case there will only be a single parity error which is not a bit error (that would require at least two parity bits to be in error) so a parity bit is in error.

How do we know whether there has been an error at all. Produce a master parity bit PM which is parity for all the message bits and the parity bits. If that shows an error then the error-correcting mechanism is invoked. OK, so what if PM is in error? - then the error-checking will show all parity bits to be OK, a pointer will be generated which has a null value and nothing will be corrected.

Those of you with time and energy might like to consider what happens for a double-bit error. (Double-bit errors can be detected but not corrected in this system).

Practical Systems

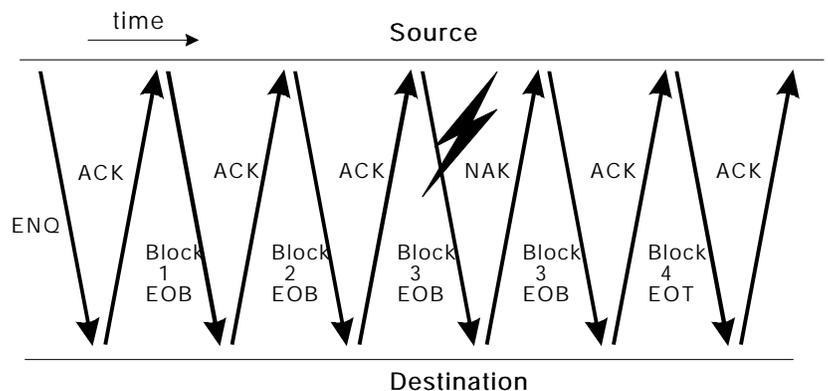
So far we have only discussed the one-way transmission of data but the possibility of bi-directional data transmission exists with a reverse channel . The commonest protocols for this are variants of the "ENQ-ACK" approach typified by the IBM 2780 protocol. In this system starts at an idle state which is terminated when either side sends an "ENQ" message whose meaning is "I want to send to you". If the other end can receive data it sends an "ACK" and then the first side sends the first data block. If the block is the total message it ends in "EOT" (end-of-transmission) or if more is to come the end is "EOB" (end-of-block). If the first block is received correctly the receiver sends an "ACK" and the next block (if any) is sent and so on. If an error is detected then a "NAK" (negative acknowledge) is sent and the block is retransmitted.

In addition to the above simple rules which apply when everything is approximately working, there are some other rules which need to be invoked in more serious circumstances:

Timeouts need to be implemented so that a reply is expected from the other end of the system in a finite time or some error action is taken (error action could be a re-send of the last block or an abort of the whole transmission). Timeouts could be caused by a sudden loss of line integrity (back-hoe through the cable!) or a system failure at the other end (power cut) etc, etc.

Collisions The only collision one can think of in this system is that both ends decide to request a message transmission at once. Under these circumstances somebody has to be the "designated winner".

The accompanying diagram shows a message being transmitted from "source" to "destination". Notice the difference between "EOB" for

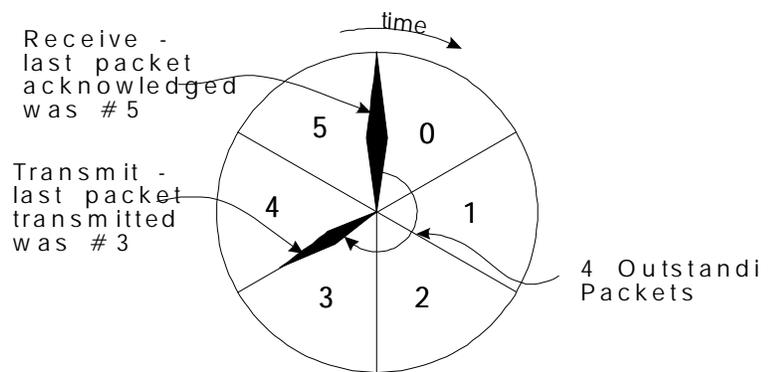


ENQ-ACK Protocol Using Half-Duplex Lines

ending intermediate blocks and "EOT" which ends the last block of the message. There is also a re-transmission of the third block due to a line error.

Notice that in this protocol only one side transmits at a time. This is called a "half-duplex" protocol and can be implemented on a one-way line if the line direction can be reversed at the appropriate moments. However most of the time lines are now "full-duplex" in the sense that information can be sent both ways at once. Sometimes the same set of wires is used in both cases and sometimes there is one set for each direction. A full-duplex protocol is possible with the above system but life gets more complicated.

Further complication is introduced if the communication link is very extended, particularly if several links are placed in series, as it takes an inordinate length of time for the "ACK" to get back. To combat this problem protocols exist for having a number of "messages" or "packets" outstanding at any one time and having separate "ACKS" for each, e.g. if there are 8 acks "ACK0"- "ACK7" it is possible for the transmitter to send out 8 packets before it needs to see an "ACK" at all. Of course if it then receives "ACK0" "ACK1" "NAK2" it must decide what to do about 3-7 but this can be handled in the (complicated) protocol rules for this system. DATAPAC, the Canadian packet-switching system uses this type of protocol.



Multiple Outstanding Packets

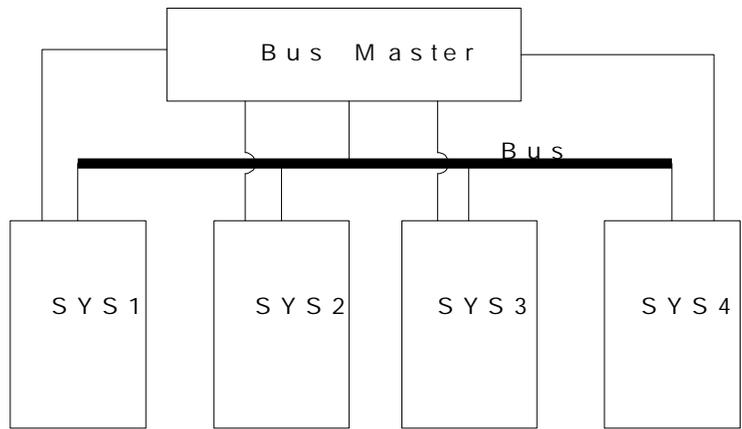
Network Communications

I discussed in the previous section some very simple ideas concerning the error-free communication between two systems at the opposite ends of a set of wires. Most of the systems dealt with were of the message/reply type where each message or block is acknowledged individually. However this gets more complicated if there are more than two systems on the lines at once - as anybody who has ever had a crossed telephone line will attest!

Protocols for networks, or bus systems for there is really not that much difference between the two, have to deal with this additional level of hassle.

The simplest system for dealing with this is to have a "bus controller" to which everybody submits their requests for data transfer and then the controller allocates the time according to the requests submitted. This works quite well on a simple computer backplane but fails in an extended network for three reasons:

Firstly because the messages to the controller must move along a different path from the main link, which means additional complication. Secondly time delays can get unreasonable. Thirdly there is one thing on the bus which is different from all other things - the bus controller.

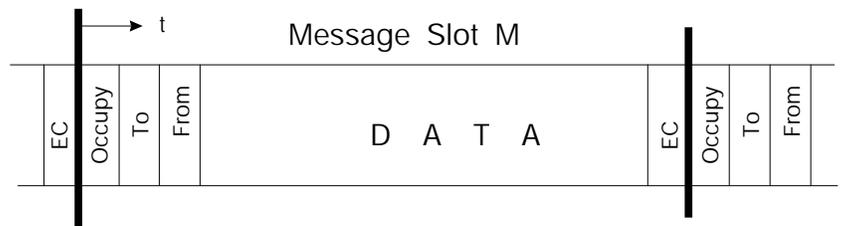


BusMaster Controller Schematic

Despite these shortcomings this system is widely used, and in fact is the basis of most "direct-memory access" systems in computers, which we shall discuss later. However that application is one in which there already is a "boss" system and the other disadvantages do not apply. Some extended systems also work on this principle - with variations.

The major problem in all these systems is how to persuade a device to put data onto the "bus" or "network" without interfering with everybody else. Receiving data is simple because one can "eavesdrop" on the bus without upsetting everybody else but transmitting is fatal if more than one system does it simultaneously. There are also hardware problems concerned with getting new signals onto the bus without upsetting the ones already there, and that is another story.

The simplest division of resources is to

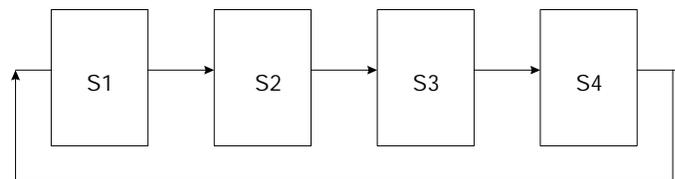


Time-Slot Message Allocation

allocate each transmitter a "message slot" on a regular basis - defined by some timing sequence on the bus and allow it to slot in (or not slot in) a message only at that time. Each message requires an address for the recipient, a "from address" for the error-check reply, the message and check information. All receivers check all messages for their address and reply to sender. This actually works quite well but is wasteful of system resource in that if there are 1024 devices on the line and only two are actually communicating, they only get 1/1024 of the time and 1023/1024 is wasted. Therefore these schemes work best in applications where the bus is continually fully loaded by communications. Since "collisions" are impossible on the bus, the fully-loaded situation tends to be more efficient for a time-slot system than any other.

If the time-slot scheme is abandoned then an allocate-on-request scheme can be tried whereby the bus stream is checked for a free message slot and then that one is filled. If you think about this then this requires that the system actually alters the allocation flags in the data block - which also implies that the bus is intercepted and delayed. The reason for this is that the input circuitry must be able to detect the "free slot" flag and by the time that has been detected it is too late to insert a "full slot" flag without delaying the whole bus slightly. This constrains the system to a ring architecture. The immediate problem that arises is that a failure in a "node" can bring the whole system down. However it does convey the benefit of good bus utilisation. Another problem is that of "lockout" whereby a device on a heavily loaded bus cannot get a look-in - rather like trying to get onto the QEW in the rush hour.

If "S1" in the diagram is talking to "S4" and using all the bus bandwidth (all available message slots) then "S2" will never find a free message slot to send anything to "S3". Therefore communications are locked until "S1" releases the line. You can easily see that in a more complex system even with some "good behaviour" rules, things could get awkward.



Ring System

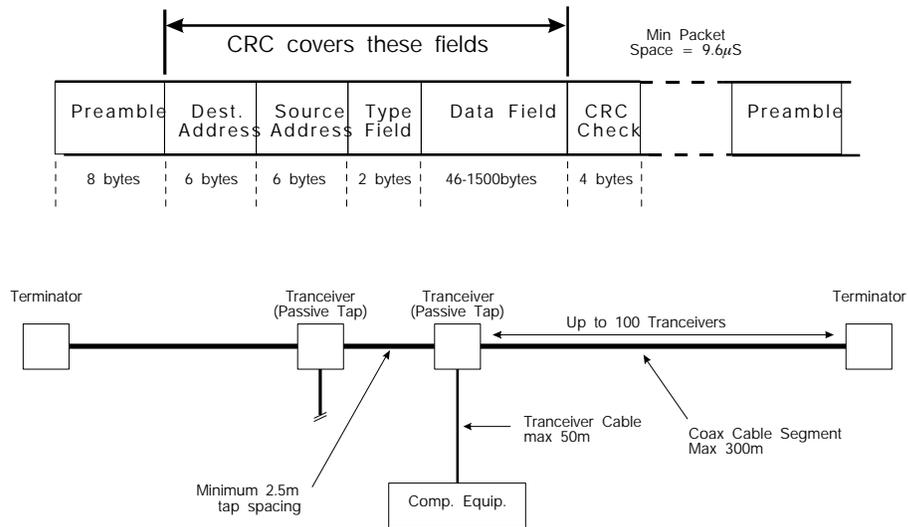
If we abandon all allocation we can institute a free-for-all (QEW in the rush hour?) which will have the benefit of not requiring a referee but needing some sophisticated rules and systems to cope with the injuries. In these systems known as "Aloha" systems (from the Hawaiian islands) anybody with a message to send first checks to see if there is any bus traffic and then starts transmitting if nobody else is. Under this scheme of course there is

a chance that two people will decide to go at once and a collision will occur. It is therefore imperative that an acknowledge-type protocol is used so that the sender can "timeout" if no reply is received from the receiver after a fixed time (total message garbling) and resend on receipt of a "NAK". Of course if two senders collide, timeout after the same time, wait a fixed (same) time and then resend, the collisions will recur indefinitely. So some "randomisation" has to be put in to prevent recurring collisions. Under this scheme everybody gets a look in but there are problems of bus usage as the system usage rises because of the inherent slowness of ACK/NAK and the increasing inefficiency due to bus collisions. Much thought has been given to overcoming such problems in systems such as "Ethernet" where many devices share one piece of co-ax cable.

Ethernet

Like other communications systems, that which is commonly referred to as "Ethernet" comprises a plethora of hardware, communication and software standards.

The hardware standard is based on "IEEE 802.3" and covers the hardware and basic communication areas. The hardware usually operates on a 50 Ω cable (although twisted-pair cable and optical fibre are also used) which is of two types "thick" and "thin". Both operate in a similar manner but the hardware method of attaching to the systems varies. Connections are required to have the property of being "passive" when they are not transmitting (including the time when they are not powered). Transmission of bits is by Manchester encoding which places a bit transition in the middle of each cell and is similar to the "clock and data" example which is given later in this volume. The data rate is 10Mbps. At this rate and with a large number of "taps" on the line, the control of line reflections and impedances becomes important and rules such as: minimum tap spacing of 2.5m and 50 Ω cable termination on both ends become important.



Ethernet Packets and Cabling Specifications

Data are transmitted in "packets" each of which has the following format:

Preamble	8bytes
Destination address	6bytes
Source address	6bytes
Type field	2bytes
Data field	46-1500bytes
Cyclic redundancy Check (CRC)	4bytes

Inter-packet gap 9.6µS.

The minimum packet length is therefore 72bytes and the maximum 1526bytes.

Communication using an Ethernet system is controlled by a higher level software protocol and one such is TCP/IP (Transmission Control Protocol/Internet Protocol) which allows communications for terminal emulation (TELNET) and file transfer (FTP). A discussion of this protocol which is the basis for our laboratory network and all our campus communications (with the exception of some DEC machines) is a bit beyond the scope of this course.