LINUX VOICE
CODING TUTORIAL

ERROR DETECTION WITH HAMMING CODES AND CRCs

BEN EVERARD

Data errors are a fact of life, but with a little clever code, you can catch them out before they trip you up.

When sending data across a network or storing it on disk drives, there's a chance that the data will become corrupted. Good hardware design can help minimise this, but it can never eliminate it completely, so it's important to be able to detect when these errors happen so that the data can be re-sent.

The easiest way to check for errors is to add a parity bit. This is an extra bit of data added after each small block that's used to detect errors. There are two types of parity: odd and even. In odd parity, the bit is used to make it so there is an odd number of 1's in the data, while even parity makes it an even number of 1's. When checking the data, the computer just counts the number of 1's and sees if it matches with the type of parity being used.

For example, the letters LV are 01001100 01010110 in ASCII. With an even parity bit added to each letter, they become 010011001 010101100. This adds one bit for every eight bytes of data, so there is more to send; however, in many cases it'll be a good trade off because there'll be fewer errors. This extra data is known as redundancy because it doesn't add any information (we still only have the letters LV, but we've sent 18 bits rather than 16).

"A parity bit is an extra bit of data added after each small block, which is used to detect errors."

A parity bit won't detect every error. If any single bit gets flipped then it will flag it up, but if two bits get flipped, it won't detect the error. It will detect an odd number of errors in the data, but not an even number of errors. You could increase the error detection by using a parity bit on a smaller piece of data. For example, you could split the above data into four-bit chunks and add a parity bit on these. This will improve the error detection, but at the expense of extra redundancy. The best trade off between the these two opposing forces will depend on how error-prone the communications channel is.

Error correction

There are other ways than just splitting the data up into ever smaller chunks. One common way is to add three parity bits for every four bits on data. In this case, the parity bits (p1, p2 and p3) are interspersed so that they only cover some of the four data bits (d1, d2, d3 and d4).

- **p1** is calculated based on even parity with data bits d1, d2 and d4.
- **p2** is calculated based on even parity with data bits d1, d3 and d4.
- **p3** is calculated based on even parity with data bits d2, d3 and d4.

The data is sent in the order p1,p2,d1,p3,d2,d3,d4.

By interlacing the parity bits in this way, you can not only tell that an error occurred, but which bit it occurred in. For example, if p1 and p2 are incorrect, but p2 is correct, then you know that the error must be in d2 (or there is more than one error). This means that if you are reasonably confident that the communications channel won't flip more than one bit out of every 7, you can correct any single-bit error. On the other hand, if it's a noisy channel, you can detect any two-bit errors (these would be corrected incorrectly if you dealt with them as single-bit errors).

Stop: Hamming time

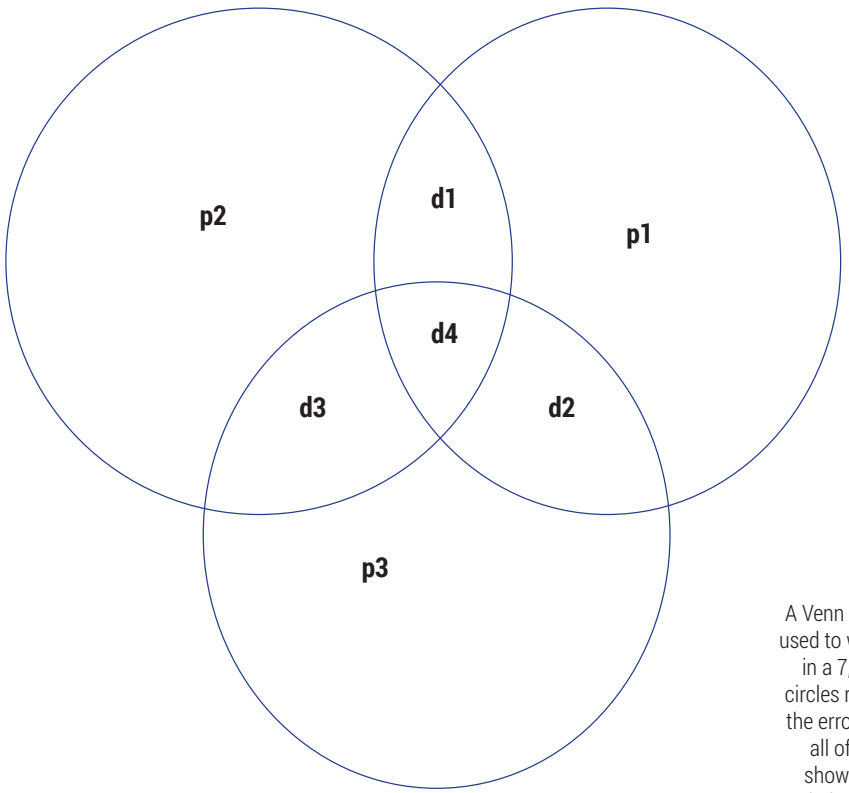
This is known as a 7,4 Hamming code, because the method was developed by Richard Hamming, and for every 7 bits set, 4 are data.

The reason we know how many bits can be corrected or detected by a particular code is because of the Hamming distance. This is the number of bits that have to flip to get from one valid sequence to another. For example, using the 7,4 hamming code, 1111 and 0111 are encoded as:

```
1,1,1,1,1,1,1
0,0,0,1,1,1,1
```

You can see that at least three bits have to flip to get from one to the other. If you calculate 7,4 Hamming codes for all possible values of d1,d2,d3 and d4, you'll find that this is the minimum number of bits different between any two, so we can say that it has a minimum Hamming distance of three. Whenever there is a single-bit error, it will be closer to one code than the other. However, if there are two-bit errors, then it will be closer to the wrong code.

If you want to detect or correct more errors, then you need to increase the Hamming distance. Again, this means increasing the redundancy. It turns out that this is easily done by adding a parity bit that covers every bit (p1–p3 and d1–d4). The result is a code that has four bits of data for every eight bits sent and a minimum Hamming distance of four (this is an 8,4 Hamming code). This can still only correct a single-bit error, but it can detect three bit errors in a



A Venn diagram like this one can be used to work out which bit is in error in a 7,4 Hamming code. The three circles represent the parity bits, and the error is the data bit that's within all of the circles of the parity bits showing errors, but not inside the circle of the parity bit not showing errors (if there is one).

single block. A 9,4 Hamming code would be needed to correct two-bit errors. It is possible to go on building codes with more and more redundancy that can detect and correct more and more errors. However, the more bits you add, the more you have to send, and so the less data you can get through the channel.

Cyclic codes

Hamming codes are good for detecting errors in very noisy channels, but other times we want a solution that's very quick to calculate and that doesn't have too much redundancy. In this case, we may use Cyclic Redundancy Checks (CRCs). The mathematics of why these work is a little complex, but in practice, they are a little like long division using XOR, and the remainder is the redundancy used to check that the received value is correct. Let's look at an example using the coefficient 1011 on the data 01001100 (the letter L in ASCII) and a three-bit CRC.

```
01001100 000 ◄— data (trailing 0s are the 3 bits for the CRC)
 1011        ◄— divisor aligned with the first 1
00010100 000 ◄— result of XOR
  1011       ◄— divisor aligned with the first 1
00000010 000 ◄— result of XOR
      10 11  ◄— divisor aligned with the first 1
00000000 110 ◄— result of XOR
```

The CRC finishes because the result is all zeros except for the three bits that form the CRC value (110).

To reverse the CRC, the full data is run through the same process, except that the CRC value takes the place of the trailing zeros. At the end, if the remainder is 0 then the CRC data is correct; otherwise there is an error in the data (we'll leave it up to you to perform this and check that the above value is correct).

CRCs don't help you fix the error in the same way that Hamming codes do, they just try to spot them. CRCs can also be designed to be longer or shorter, and the length will make them better or worse at finding errors.

The reason this is very efficient to implement in hardware is that it can be done with little more than a shift register and a few logic gates. This is far less than what is needed for Hamming codes. Because of this simplified requirement, it can run in real time even on very fast data transfer, and you'll find CRCs in everything from CDs and DVDs to Ethernet and 3G mobile networks.

The outside world is a scary place for data. There is all manner of electromagnetic interference and other problems just waiting to flip 0s into 1s. With a little careful planning, though, we can protect our precious digits and make sure they don't come to any harm.

Ben Everard is the best-selling co-author of the best-selling *Learning Python With Raspberry Pi*.