**LSP 121**

**Activity 16**

**Fixing Errors**

We have already discussed the concepts of binary number systems and ASCII. Since it is important to understand these before we go on, let’s do a quick review.

Each bit position is associated with a value (a power of 2):

 64 32 16 8 4 2 1

1. If the filled circle means “count me”, then what is the decimal value of the following:

2. What is the corresponding binary value? (Just write the 1s and 0s in place of the filled/unfilled circles.)

3. Using the following ASCII table, what are the 7-bit ASCII values for the characters C D M?

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Space | 0100000 | 0 | 0110000 | @ | 1000000 | P | 1010000 | ‘ | 1100000 | p | 1110000 |
| ! | 0100001 | 1 | 0110001 | A | 1000001 | Q | 1010001 | a | 1100001 | q | 1110001 |
| “ | 0100010 | 2 | 0110010 | B | 1000010 | R | 1010010 | b | 1100010 | r | 1110010 |
| # | 0100011 | 3 | 0110011 | C | 1000011 | S | 1010011 | c | 1100011 | s | 1110011 |
| $ | 0100100 | 4 | 0110100 | D | 1000100 | T | 1010100 | d | 1100100 | t | 1110100 |
| % | 0100101 | 5 | 0110101 | E | 1000101 | U | 1010101 | e | 1100101 | u | 1110101 |
| & | 0100110 | 6 | 0110110 | F | 1000110 | V | 1010110 | f | 1100110 | v | 1110110 |
| ‘ | 0100111 | 7 | 0110111 | G | 1000111 | W | 1010111 | g | 1100111 | w | 1110111 |
| ( | 0101000 | 8 | 0111000 | H | 1001000 | X | 1011000 | h | 1101000 | x | 1111000 |
| ) | 0101001 | 9 | 0111001 | I | 1001001 | Y | 1011001 | i | 1101001 | y | 1111001 |
| \* | 0101010 | : | 0111010 | J | 1001010 | Z | 1011010 | j | 1101010 | z | 1111010 |
| + | 0101011 | ; | 0111011 | K | 1001011 | [ | 1011011 | k | 1101011 | { | 1111011 |
| ‘ | 0101100 | < | 0111100 | L | 1001100 | \ | 1011100 | l | 1101100 | | | 1111100 |
| , | 0101101 | = | 0111101 | M | 1001101 | ] | 1011101 | m | 1101101 | } | 1111101 |
| . | 0101110 | > | 0111110 | N | 1001110 | ^ | 1011110 | n | 1101110 | ~ | 1111110 |
| / | 0101111 | ? | 0111111 | O | 1001111 | \_ | 1011111 | o | 1101111 |  | 1111111 |

What if we wanted to send the text message “cool” to a friend (in response to something they sent you). Your cell phone would convert each character to ASCII, giving you the following:

 1100011 1101111 1101111 1101100

Unfortunately, there is always some amount of noise on a transmission line when you are talking / texting / surfing the Internet / downloading a song / sending someone a picture from your phone. In fact, in the networking business, they jokingly say there are three constants in the universe: death, taxes, and noise. So what would happen if noise messed up the first letter in your text message? More precisely, what happens if one bit in the first character is altered, resulting in the following message being delivered?

 1100010 1101111 1101111 1101100

4. What message was actually delivered?

I suspect your friend would probably text back saying something like “what planet are you from?” In order to catch transmission errors, transmission systems add some extra bits to your message. There are a number of techniques used today, but let’s take a look at a simple one called *parity*. To perform parity, an 8th bit is added to the end of each 7-bit character. How do we know if this added bit is a 0 or a 1? The transmitter counts the number of binary 1s in the character, and if this sum is an even number of 1s, then a parity bit of 0 is added. Notice that by adding a 0, we still have an even number of 1s. If there are an odd number of 1s in the character, then the transmitting device adds a parity bit of 1. By adding a 1, we once again have an even number of 1s in the character. For example, if we wanted to send the character 1100001, we would add another 1 to the end of the character, thus preserving an even number of 1s. The transmitted character would thus look like 11000011.

5. What is the parity bit that is added to the following character: 1011010

6. What is the parity bit that is added to 1101101?

Let’s assume the transmitter wants to send the letter “c”, thus it will send 1100011. The parity bit of 0 is added to the end, giving us 11000110. The text message is sent and noise alters the 7th bit, giving us 110001*0*0. The receiver, knowing that the eighth bit is a parity bit, will add up the 1s in the character. If there is an even number of 1s, there is no error. But if there is an odd number of 1s, there is an error.

7. If the character 11000100 is received, is there a parity error?

Parity checks are not the only way to identify errors. And the errors don’t always involve computers. Swiss and German railways use a special system to reduce errors when locomotive numbers are recorded. On each of these locomotives you would find a 7-digit train number. The first six digits were the actual train number while the 7th digit was a check digit. The check digit has been chosen to reveal any errors in typing in the train number. To demonstrate how this works, you do the following: Add together the 1st, 3rd, 5th and 7th digits. For example, if the train number is 162-943-7, add up 1 + 2 + 4 + 7, giving 14. Then add together the 2nd, 4th, and 6th digit, and double this sum. Again using 162-943-7, add up 6 + 9 + 3, giving 18, then double, giving 36. Now add the two sums together: 14 + 36 = 50. If the final sum is a multiple of 10, then chances are pretty good that the train number was recorded correctly. If not a multiple of 10, someone recorded the number incorrectly.

8. Given the train number 153-524-6, was this number recorded correctly? Show your work.

9. Given the train number 134-824-3, was this number recorded correctly? Show your work.

This type of error detection is called *arithmetic checksum* and is used extensively on the Internet. It is also used on things like credit card numbers, travelers cheques, airline tickets, Euros paper money, Universal Product Codes (UPC), and ISBN numbers (for books).

For example, let’s take a closer look at UPCs. Consider the number 0 38000 00127 7 found on a box of cereal. The first digit identifies a broad category of goods, the next five digits identify the manufacturer, the second five digits identify the product, and the last digit is a check digit. Take any UPC number such as a1 a2 a3 a4 a5 a6a7a8a9a10a11a12. To calculate the checksum, perform the following calculation: 3a1 + a2 + 3a3 + a4 + 3a5 + a6 + 3a7 +a8 + 3a9 + a10 + 3a11 + a12. The sum should end with a 0. If it doesn’t, then someone entered an incorrect UPC. This error detection method catches all single-position errors and 89% of all other kinds of errors.

10. Is the following UPC correct or incorrect? 3 24500 00054 8

How about a checksum method that catches all errors? The ISBN number that is on the back of all books can detect 100% of single errors and 100% of transposition errors. To perform the checksum on the 10-digit ISBN a1a2…a10, calculate: 10a1 + 9a2 +8a3 + 7a4 + 6a5 + 5a6 + 4a7 + 3a8 + 2a9 + a10 and then divide this total by 11. If it divides evenly, there is no problem. If it doesn’t divide evenly, there is an error.

11. Is the following ISBN code correct or incorrect? 1-133-62646-7

Let’s go back to parity errors. If there is a parity error, and without knowing what was originally sent, does the receiver know what the character should have been? No. There are too many other characters one bit different from what was received to tell what was actually sent. For example, let’s say we want to transmit the character 1011001. We add a parity bit of 0, giving us 10110010. During the transmission, one bit is messed up and the following is received: 10100010. When the receiver does a parity check, it counts an odd number of 1s and knows there is an error. But without knowing what was originally sent, can you tell which bit was messed up? No, you can’t. Any one of the bits in the character could have been flipped. But what if we took each bit and before we sent it, we tripled each bit. If we wanted to send

10110010

 we would instead send

111 000 111 111 000 000 111 000

Now when we send that 24-bit character, one of the bits is flipped, giving us:

 111 000 111 110 000 000 111 000

Can you tell which bit was just altered? You can because every bit was tripled before sending. If one bit in a group of three bits differs from the others, then we take the majority and flip the one outstanding bit. This is called *error correction*, because the receiver, upon noticing that one bit in a group of three bits is different, flips the bit to make it the same as the other two, thus correcting the error. This is how the space shuttle operates. Many tasks on the shuttle require a computer. But what if the computer malfunctions? Lives could be at stake. So to be more careful, there are three computers to do each operation. When asked to perform a computation, all three computers should give the same result. If one computer differs from the other two, then we can take a majority. (But what happens if two computers are in error?)

A more useful type of forward error correction is a Hamming code. A Hamming code is a specially designed code in which special check bits have been added to data bits such that, if an error occurs during transmission, the receiver may be able to correct the error using the included check and data bits. For example, let us say we want to transmit an 8-bit character, for example, the character 01010101 seen in the figure below. Let us number the bits of this character b12, b11, b10, b9, b7, b6, b5, and b3. (We will number the bits from right to left, leaving spaces for the soon-to-be-added check bits.) Now add to these data bits the following check bits: c8, c4, c2, and c1, where c8 generates a simple even parity for bits b12, b11, b10, and b9. The check bit c4 will generate a simple even parity for bits b12, b7, b6, and b5. Check bit c2 will generate a simple even parity for bits b11, b10, b7, b6, and b3. Finally, c1 will generate a simple even parity for bits b11, b9, b7, b5, and b3. Note that each check bit here is checking different sequences of data bits.



Let us take a closer look at how each of the Hamming code check bits in the above figure works. Note that c8 “covers” bits b12, b11, b10, and b9, which are 0101. If we generate an even parity bit based on those four bits, we would generate a 0 (there are an even number of 1s). Thus, c8 equals 0. c4 covers b12, b7, b6, and b5, which are 0010, so c4 equals 1. c2 covers b11, b10, b7, b6, and b3, which are 10011, so c2 equals 1. c1 covers b11, b9, b7, b5, and b3, which are 11001, so c1 equals 1. Consequently, if we have the data 01010101, we would generate the check bits 0111, as shown in the figure. This 12-bit character is now transmitted to the receiver. The receiver accepts the bits and performs the four parity checks on the check bits c8, c4, c2, and c1. If nothing happened to the 12-bit character during transmission, all four parity checks should result in no error. But what would happen if one of the bits is corrupted and somehow ends up the opposite value? For example, what if bit b9 is corrupted? With the corrupted b9, we would now have the string 010000101111. The receiver would perform the four parity checks, but this time there would be parity errors. More precisely, because c8 checks b12, b11, b10, b9, and c8 (01000), there would be a parity error—as you can see, there are an odd number of 1s in the data string, but the check bit is returning a 0. c4 checks b12, b7, b6, b5, and c4 (00101), and thus would produce no parity error. c2 checks b11, b10, b7, b6, b3, and c2 (100111), and would produce no parity error. c1 checks bits b11, b9, b7, b5, b3, and c1 (100011), which would result in a parity error. Notice that if we examine just the check bits and denote a 1 if there is a parity error and a 0 if there is no parity error, we would get 1001 (c8 error, c4 no error, c2 no error, c1 error). 1001 is binary for 9, telling us that the bit in error is in the ninth position.

Despite the additional costs of using forward error correction, are there applications that would benefit from the use of this technology? Two major groups of applications can in fact reap a benefit: digital television transmissions and applications that send data over very long distances. Digital television signals use advanced forms of forward error correction called Reed-Solomon codes and Trellis encoding. Note that it is not possible to ask for retransmission of a TV signal if there is an error. It would be too late! The data needs to be delivered in real time, thus the need for forward error correction.

12. Let’s try one. A system is going to transmit the byte 10010100. Show the four check bits c8, c4, c2 and c1 that will be added to this byte. Show your work.

13. The 12-bit string 010111110010 with embedded Hamming code bits (c8, c4, c2 and c1) has just arrived. Is there an error? If so, which bit is in error? Show your work.