Chapter 2

Encoding Text with a Small Alphabet

Given the nature of the Internet, we can break the process of understanding how information is transmitted into two components. First, we have to figure out how each type of information we might wish to transmit through the network can be represented using the binary alphabet. Then, we have to learn how 0's and 1's can actually be sent through a wire. We will consider how to represent information in binary in this and the following chapter. Then, with this understanding we will look at the process of transmitting binary information in the following chapter.

It is clear that it is possible to transmit information of many forms. Images, sound, and movies are among the obvious examples. Many Internet protocols, including those used for email and text messaging, however, rely mainly on the transmission of text messages. To simplify our task, we will therefore initially limit our attention to discussing how text is encoded in binary.

Even with our attention limited to encoding text, working in binary can be painful. Fortunately, we can postpone the ordeal of working with binary while grasping most of the principles behind binary encodings by first considering the problem of how we might encode text using only the ten digits 0 through 9. Then, we can apply the understanding we gain about such encodings to the less familiar world of binary.

2.1 Doing without Delimiters

You can easily represent letters from the alphabet using only digits by simply numbering the letters. For example, the table below shows an obvious way to number the letters of the alphabet:

1. a	11. k	21. u
2. b	12. l	22. v
3. c	13. m	23. w
4. d	14. n	24. x
5. e	15. o	25. y
6. f	16. p	26. z
7. g	17. q	
8. h	18. r	
9. i	19. s	
10. j	20. t	

Given this table, we can represent any word by listing the numbers for each of the letters in order. For example, "bad" could be encoded as "214." In a scheme like this, the sequences used to represent individual letters are called *codewords*. That is, "2" is the codeword for b and "18" is the codeword for r. A sequence of codewords like "214" that is intended to represent a complete message will be called a *coded message*.

The simple scheme described by the table above runs into trouble very quickly if we try encoding something just a bit more complicated like the word "barn". The coded message derived from our table for "barn" would be "211814" since b = 2, a = 1, r = 18 and n = 14. Unfortunately, this is also the coded message for "urn" since u = 21, r = 18 and n = 14. If we were actually using a scheme like this to send text through a network, the computer (or person) on the other end would be confused if the coded message "211814" arrived. The problem here is that there is no way for the receiver to know whether to interpret "21" as a "2" representing "b" followed by a "1" representing "a" or as the pair "21" representing "u".

We might think of fixing this by separating codewords that represent distinct letters from one another with commas so that barn would be represented by "2,1,18,14" and urn would be represented by "21,18,14". If we do this, however, we are no longer representing text using just the 10 digits. We are now using an eleventh character in our scheme, the comma. It may not seem like a big issue to use 11 symbols rather than 10, but remember our ultimate goal is to use the techniques we explore using the 10 decimal digits to understand encoding schemes based on binary digits. The whole

point of binary is to get down to the smallest useful alphabet, the alphabet with only two symbols. If one of the two symbols used in that case is a comma, we really only have one symbol to work with. So, we need to learn to do without commas now!

You might get the clever idea that we can accomplish the purpose of the commas without actually using an extra symbol by simply leaving spaces between the codewords where the commas would have appeared. Thus, barn would be represented by "2 1 18 14" and urn would be represented by "21 18 14". Alas, this doesn't really solve the problem. What it does instead is point out that the space is as significant a symbol as the comma is. Although they don't use up any ink, spaces definitely convey information. Consider the two sentences:

> "I want you to take him a part."

and

> "I want you to take him apart."

Spaces in text are symbols just like the letters of the alphabet. If we use spaces to separate the digits as in "21 18 14" we are again using an 11 symbol alphabet rather than a 10 symbol alphabet.

Fortunately, there are approaches that will enable us to tell barn and urn apart without using anything other than the 10 digits. One simple technique is to fix the number of digits used to represent letters from the original alphabet. The problem in our original scheme is that some letters from the alphabet are encoded as a single digit while others require 2 digits. The problem goes away if we use two digits for every codeword. We can do this by starting the numbering of the alphabet with a two digit number (i.e., a = 10, b = 11, c = 12, etc.) or even more simply by adding a leading zero to each single digit number (i.e. a = 01, b = 02, etc.). With this change, barn becomes "02011814" while urn is represented as "211814". Thus, by choosing some fixed number of digits to represent each symbol in the alphabet, we can avoid the need to use any delimiters in our representation scheme.

2.1.1 How Many Digits

If we avoid using delimiters by fixing the number of digits used to represent each symbol of the alphabet, we have to decide how many digits to use. Clearly, 1 digit would not be enough. It would only enable us to encode 10 distinct letters. On the other hand, 2 digits seems sufficient. With 2 digits

we can encode 100 distinct symbols and there are only 26 symbols in the alphabet.

If we want to use our encoding scheme to represent the contents of real messages, of course, we will have to be prepared to represent more than the 26 lower case letters. We will certainly have to handle upper case letters. We could do this by numbering the upper case letter with the next 26 numbers so that A = 27, B = 28 and so on. Then, of course, there are the digits themselves. If someone typed the number "4" in an e-mail message, we could not just encode it as "4". First, our scheme depends on using exactly two digits for each character. So, we have to use a pair of digits to encode the single character "4". Moreover, given the encoding rules we have proposed so far, we could not use "04" to encode "4" because "04" already encodes "d".

The simplest alternative is to number the digits as we numbered the letters starting with the first number that hasn't yet been used for an upper or lower case letter, 53. So, we would represent 0 as "53", 1 as "54", 2 as "55" and so on. Of course, we could start over again and re-number the letters starting at 10 so that we could use "00", "01", "02", ... and "09" for the digits. This might seem more "natural" to us, but it wouldn't be superior to the other scheme in any significant way.

Next, we have to worry about punctuation marks that might appear in the text. Things like commas, quotes, semi-colons and question marks certainly need to be included. Also, just as we discovered we had to think of spaces as symbols if we tried to use them as delimiters, we better provide a way to encode the spaces that appear between words in messages.

If it still troubles you to think of the space as a character that is as important as things like "e"s and periods, think a bit about how an encoding scheme like ours might actually be used in a computer. When you press a key on your keyboard, the electronic components in the keyboard have to send some sort of message through the cable connecting the keyboard to the computer to tell the computer what character was typed. These messages are numbers expressed in binary. For our purposes, however, we could imagine that they were expressed using the scheme we are developing. That is, when one typed a "c" on the keyboard, the keyboard might send the sequence "03" to the computer. Clearly, for such a system to work, the keyboard needs some message it can send when you press the space bar. Similarly, it needs to send messages informing the computer when you press the return or tab key. So, in addition to the normally recognized punctuation marks, any scheme for encoding text in a computer must include encodings for characters like the space, tab and return.

Looking at the keyboard in front of me I see that the combination of punctuation marks, the space key, etc. account for about 36 additional symbols. Together, the 52 alphabetic character, the 10 digits and the punctuation marks account for about 98 characters that need to be encoded.

This should make you nervous.

Recall that if we use 2 decimal digits to encode each symbol, we can encode up to 100 distinct symbols. At this point, we are already using all the pairs up to about 98. There are only 2 left: 99, and 00. Basically, there isn't much room left for expansion.

Suppose someone wants to design a new keyboard that provides more characters. For example, while my keyboard includes the "$", it does not include the symbol for the British pound, £. It is also missing the section symbol, §, and the copyright symbol, ©. If the new keyboard is to include more than 2 such additional characters, our code must be revised. If we wanted to be able to encode more than 100 distinct characters, we would have to use 3 digits for each codeword rather than just 2. We need not change the values of the numbers associated with symbols in our original scheme, but each character would need to be encoded using three digits. Thus, "a" would be encoded as "001" rather than as "01", "b" would be encoded as "002", and so on. With this scheme, we could encode up to 1000 distinct characters.

To appreciate the impact of making such a change in a coding scheme, consider again how such a code would be used. Computer keyboards might use the code to send signals to an attached computer when a key is pressed. The connection between the keys on the keyboard and the digits sent will be built into the hardware of the keyboard. If it later became necessary to change the code, the keyboard would have to be discarded and replaced with a new keyboard designed to use the new code. Chances are, in fact, that the computer would be similarly dependent on the code and have to be discarded with the keyboard. Thus, even though we can easily think up many different codes for keyboard symbols, it isn't easy to change from one to another. In the real world, leaving room for expansion may make it possible to extend the code later without requiring costly hardware replacements.

2.2 Moving from Decimal to Binary

Encoding text in binary is fundamentally the same as encoding text using decimal digits. The simplest approach is again to picked a fixed number of binary digits to use for every character. It is still important to choose enough digits to leave room for expansion. The big difference is that there are only two distinct digits in binary, 0 and 1. As a result, longer codewords are required to represent each character. While we saw that 3 decimal digits would be sufficient to represent all of the characters used when writing English text leaving plenty of room for expansion, we will see that 7 or 8 digits are required in binary.

Suppose, for a moment, that we did try to get away with just 2 binary digits. We saw that 2 decimal digits would be enough to encode up to 100 distinct characters. How many characters can 2 binary digits encode? The short answer is not many! If you start writing down all the combinations of pairs of binary digits you can find, you will run out after writing just four pairs: 00, 11, 01, 10.

Even if you allow yourself to use 3 binary digits, the collection of possibilities doesn't get much bigger. In particular, with 3 digits the only combinations are 000, 011, 001, 010, 100, 111, 101, and 110.

Writing down all the possibilities for longer sequences of binary digits would be painful. Luckily, there is a simple rule at work. There were four 2 digit binary sequences and eight 3 digit binary sequences. Allowing an extra digit doubled the number of possibilities. If we allowed a 4th digit, we would find there were again twice as many possibilities giving 16. In general, if we use N binary digits, we will have enough combinations to represent 2^N distinct symbols. Therefore, with 6 binary digits, we could handle $2^6 = 64$ symbols. This is fewer than the 98 symbols on my keyboard. With 7 binary digits we could handle $2^7 = 128$ symbols. This will handle the 98 symbols found on my keyboard and leave a reasonable amount of room for expansion. In fact, the code that is actually used to represent text characters on most computers uses 7 binary digits.

The code used to represent text in most computers and network messages today is called ASCII, which stands for "American Standard Code for Information Interchange." It is very much like the decimal code we described above. The letter "a" is represented as 1100001, which is the binary form for the decimal number 97. The letter "b" is represented by 1100010, which corresponds to 98. The remaining letters are associated with consecutive binary numbers. Capital letters work similarly with "A" represented by 1000001, the binary equivalent of 65. For those who really want to know

more, a list of ASCII codes can be found in Figure 2.1.

It is common to add one additional digit to the sequences of 7 binary digits used to represent symbols by the ASCII code. The value of this extra digit is chosen in a way that makes it possible to detect accidental changes to the sequence of digits that might occur in transmission. This extra digit is called a parity bit. We will discuss parity bits in more detail in a later chapter. For now, the main point is that characters encoded in ASCII actually occupy 8 binary digits of computer memory.

Another widely used code for representing text in binary is called EBCDIC (for Extended Binary Coded Decimal Interchange Code). It was developed by IBM and used as the standard code on several series of IBM computers. When IBM mainframes dominated the computing world, EBCDIC, rather than ASCII, was the most widely used text encoding scheme. EBCDIC differed from ASCII in many ways. The same sequence of digits that represented "z" in ASCII was used to encode the colon in EBCDIC. They did, however, use a similar number of binary digits to encode characters. In EBCDIC each character was encoded using 8 binary digits.

The encoding of text data is quite important in computing and computer networking. It is important enough that the unit of memory required to encode characters of text in these common codes is also used as the standard unit for measuring memory. The actual memory of a computer is composed of millions of binary digits. The term *bit* is used to refer to a single binary digit. The hardware of most machines, however is designed so that the smallest unit of memory that can be easily, independently accessed by a program is a group of 8 bits. Eight bits is enough to hold one character in either of the most widely used text encoding schemes. Such a group is called a *byte*.

Before leaving the subject of encoding text using fixed-length binary codewords, we should mention one other standard for character representation, a relatively new code named Unicode. All of the examples given above have been embarrassingly ethnocentric. We have explained how to represent English text, but ignored the fact that many other languages exist, are used widely, and often use different alphabets. The 128 possible letters provided by ASCII are woefully inadequate to represent the variety of characters that must be encoded if we are to support everything from English to Greek to Chinese. Unicode is a text encoding standard designed to embrace all the world's alphabets. Rather than using 7 or 8 bits, Unicode represents each character in 16 bits enabling it to handle up to 65,536 ($= 2^{16}$) distinct symbols. For compatibility sake, the letters and symbols available using ASCII are encoded in Unicode by simply adding enough zeros to the left end of the

Binary	Decimal	Hex	Graphic
0010 0000	32	20	(blank) (□)
0010 0001	33	21	!
0010 0010	34	22	"
0010 0011	35	23	#
0010 0100	36	24	$
0010 0101	37	25	%
0010 0110	38	26	&
0010 0111	39	27	'
0010 1000	40	28	(
0010 1001	41	29	)
0010 1010	42	2A	*
0010 1011	43	2B	+
0010 1100	44	2C	,
0010 1101	45	2D	-
0010 1110	46	2E	.
0010 1111	47	2F	/
0011 0000	48	30	0
0011 0001	49	31	1
0011 0010	50	32	2
0011 0011	51	33	3
0011 0100	52	34	4
0011 0101	53	35	5
0011 0110	54	36	6
0011 0111	55	37	7
0011 1000	56	38	8
0011 1001	57	39	9
0011 1010	58	3A	:
0011 1011	59	3B	;
0011 1100	60	3C	<
0011 1101	61	3D	=
0011 1110	62	3E	>
0011 1111	63	3F	?

Binary	Decimal	Hex	Graphic
0100 0000	64	40	@
0100 0001	65	41	A
0100 0010	66	42	B
0100 0011	67	43	C
0100 0100	68	44	D
0100 0101	69	45	E
0100 0110	70	46	F
0100 0111	71	47	G
0100 1000	72	48	H
0100 1001	73	49	I
0100 1010	74	4A	J
0100 1011	75	4B	K
0100 1100	76	4C	L
0100 1101	77	4D	M
0100 1110	78	4E	N
0100 1111	79	4F	O
0101 0000	80	50	P
0101 0001	81	51	Q
0101 0010	82	52	R
0101 0011	83	53	S
0101 0100	84	54	T
0101 0101	85	55	U
0101 0110	86	56	V
0101 0111	87	57	W
0101 1000	88	58	X
0101 1001	89	59	Y
0101 1010	90	5A	Z
0101 1011	91	5B	[
0101 1100	92	5C	\
0101 1101	93	5D	]
0101 1110	94	5E	^
0101 1111	95	5F	_

Binary	Decimal	Hex	Graphic
0110 0000	96	60	`
0110 0001	97	61	a
0110 0010	98	62	b
0110 0011	99	63	c
0110 0100	100	64	d
0110 0101	101	65	e
0110 0110	102	66	f
0110 0111	103	67	g
0110 1000	104	68	h
0110 1001	105	69	i
0110 1010	106	6A	j
0110 1011	107	6B	k
0110 1100	108	6C	l
0110 1101	109	6D	m
0110 1110	110	6E	n
0110 1111	111	6F	o
0111 0000	112	70	p
0111 0001	113	71	q
0111 0010	114	72	r
0111 0011	115	73	s
0111 0100	116	74	t
0111 0101	117	75	u
0111 0110	118	76	v
0111 0111	119	77	w
0111 1000	120	78	x
0111 1001	121	79	y
0111 1010	122	7A	z
0111 1011	123	7B	{
0111 1100	124	7C	|
0111 1101	125	7D	}
0111 1110	126	7E	~

Figure 2.1: Table of ASCII codes

ASCII encoding to get 16 bits.