Chapter 2

Codes

In the previous chapter we examined the fundamental unit of information, the bit, and its various abstract representations: the Boolean bit (with its associated Boolean algebra and realization in combinational logic circuits), the control bit, and the two types of physical bits, the quantum bit and the classical bit.

A single bit is useful if only two answers to a question are possible. Examples might include the result of a coin toss (heads or tails), the gender of a person (male or female), the verdict of a jury (guilty or not guilty), and the correctness of an assertion (true or false). But most situations in life are more complicated. This chapter is concerned with how complex objects can be represented not by a single bit, but by arrays of bits.

It is convenient to focus on a very simple model of a system, shown in Figure 2.1, in which the input is one of a predetermined set of objects, or "symbols," the identity of the particular symbol chosen is encoded in an array of bits, these bits are transmitted through space or time, and then are decoded at a later time or in a different place to determine which symbol was originally chosen. In later chapters we will augment this model to deal with issues of robustness and efficiency.

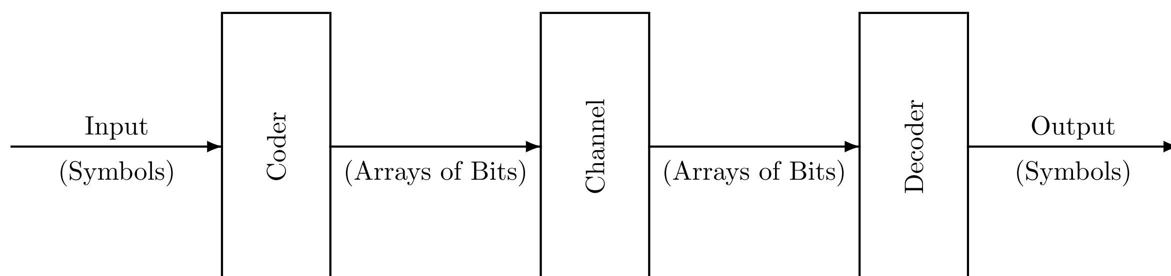

Figure 2.1: Simple model of a communication system

The basic idea behind the design of codes is simple enough: assign a different sequence of bits to each symbol being encoded. However, there are many ways to do this, some better than others. In this chapter we will look at several aspects of the art of code design, and show examples where the code was designed well or not so well.

Here are a few examples of common objects for which codes are needed, taken from computer programming, communications, and everyday life:

Author: Paul Penfield, Jr.
This document: http://www.mtl.mit.edu/Courses/6.050/2013/notes/chapter2.pdf
Version 1.7, February 4, 2013. Copyright © 2013 Massachusetts Institute of Technology
Start of notes · back · next | 6.050J/2.110J home page | Search | Comments and inquiries

- Letters of the alphabet: BCD, EBCDIC, ASCII, Unicode, Morse Code
- Integers: Binary, Gray, 2's complement
- Numbers: Floating-Point
- Proteins: Genetic Code
- Telephones: NANP, International codes
- Hosts: Ethernet, IP Addresses, Domain names
- Images: TIFF, GIF, and JPEG
- Audio: MP3
- Video: MPEG

2.1 Symbol Space Size

In designing a code, the first question to ask is how many symbols need to be encoded (the **symbol space size**). We will consider symbol spaces of different sizes:

- 1
- 2
- Integral power of 2
- Finite
- Infinite, Countable
- Infinite, Uncountable

If there is only 1 symbol, no bits are required to specify it (the symbol is already known). If there are exactly 2 symbols, the selection can be encoded in a single bit. If the number of possible symbols is 4, 8, 16, 32, 64, or another integral power of 2, then the selection may be coded in the number of bits equal to the logarithm, base 2, of the symbol space size. Thus 2 bits can designate the suit (clubs, diamonds, hearts, or spades) of a playing card, and 5 bits can encode the selection of one student in a class of 32. A dreidel is a four-sided toy marked with Hebrew letters, spun like a top in a children's game, typically at Hanukkah. The result of each spin could be encoded in 2 bits.

If the number of symbols is finite but not an integral power of 2, then the number of bits that would work for the next higher integral power of 2 can be used to encode the selection, but there will be some unused bit patterns. Examples include the 10 digits, the six faces of a cubic die, the 13 denominations of a playing card, and the 26 letters of the English alphabet. In each case, there is spare capacity (6 unused patterns in the 4-bit representation of digits, 2 unused patterns in the 3-bit representation of a die, etc.) What to do with this spare capacity is an important design issue that will be discussed in the next section.

If the number of symbols is infinite but countable (able to be put into a one-to-one relation with the integers) then a bit string of a given length can only denote a finite number of items from this infinite set. Thus, a 4-bit code for non-negative integers might designate integers from 0 through 15, but not be able to handle integers outside this range. If, as a result of some computation, it were necessary to represent larger numbers, then this "overflow" condition would have to be handled in some way.

If the number of symbols is infinite and uncountable (such as the value of a physical quantity like voltage or acoustic pressure) then some technique of "discretization" must be used to replace each possible value by one of a finite (or perhaps countably infinite) number of predefined values. For example, if the real numbers

between 0 and 1 were the symbols and if 2 bits were available for the coded representation, one approach might be to approximate all numbers between 0 and 0.25 by the number 0.125, all numbers between 0.25 and 0.5 by 0.375, and so on. Whether such an approximation is adequate depends on how the decoded data is used. The approximation is not reversible, in that there is no decoder which will recover the original symbol given just the code for the approximate value. However, if the number of bits available is large enough, then for many purposes a decoder could provide a number that is close enough. Floating-point representation of real numbers in computers is based on this philosophy.

2.2 Fixed-Length and Variable-Length Codes

A decision must be made very early in the design of a code whether to represent all symbols with codes of the same number of bits (fixed length) or to let some symbols use shorter codes than others (variable length). Both schemes have their advantages.

Fixed-length codes are usually easier to deal with because both the coder and decoder know in advance how many bits are involved. With variable-length codes, the decoder needs to tell when the code for one symbol ends and the next one begins. On the other hand, variable-length codes can lead to bit strings that are smaller on average if more frequently occurring symbols are assigned to shorter bit sequences.

2.2.1 Fixed Length

Fixed-length codes can be supported by parallel transmission, in which the bits are communicated from the coder to the decoder simultaneously, for example by using several wires, one per bit, to carry the voltages. This approach should be contrasted with serial transport of the coded information, in which a single wire sends a stream of bits and the decoder must decide when the bits for one symbol end and those for the next symbol start. If a decoder gets mixed up, or looks at a stream of bits after it has started, it might not know. This can lead to what is called a **framing error**. To eliminate framing errors, stop bits are often sent between symbols; typically ASCII (a fixed-length code) sent over serial lines has 1 or 2 stop bits, normally given the value 0. Thus if a decoder is out of step, sooner or later it will find a 1 in what it thought should be a stop bit, and it can try to resynchronize. Although in theory framing errors could persist for long periods, in practice use of stop bits works well.

2.2.2 Variable Length

An example of a variable-length code is Morse Code, developed for the telegraph. The codes for letters, digits, and punctuation are sequences of dots and dashes with short-length intervals between them. See Section 2.9.

The design of optimal variable-length codes is discussed in more detail in Chapter 5.

2.3 Use of Spare Capacity

In many situations there are some unused code patterns, because the number of symbols is not an integral power of 2. There are many strategies to deal with this. Here are some:

- Ignore
- Cry for help
- Map to other values
- Reserve for future expansion
- Use for control codes

- Use for common abbreviations

These approaches will be illustrated with examples of common codes.

2.3.1 Binary Coded Decimal (BCD)

A common way to represent the digits 0 - 9 is by the ten four-bit patterns shown in Table 2.1. There are six bit patterns that are not used (for example 1010), and the question is what to do with them. Here are a few ideas that come to mind.

First, the unused bit patterns might simply be ignored. If a decoder encounters one, perhaps as a result of an error in transmission or an error in encoding, it might return nothing, or perhaps signal some kind of error. Or it might be best to convert the invalid patterns into legal ones. For example, the unused patterns might all be converted to 9, under the theory that they represent 10, 11, 12, 13, 14, or 15, and the closest digit is 9. Or they might be decoded as 2, 3, 4, 5, 6, or 7, by setting the initial bit to 0, under the theory that the first bit might have gotten corrupted. Neither of these theories is particularly appealing, but in the design of a system using BCD, some such action must be provided.

Digit	Code
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1

Table 2.1: Binary Coded Decimal

2.3.2 Genetic Code

Another example of mapping unused patterns into legal values is provided by the Genetic Code, described in Section 2.7. A protein consists of a long sequence of amino acids, of 20 different types, each with between 10 and 27 atoms. Living organisms have millions of different proteins, and it is believed that all cell activity involves proteins. Proteins have to be made as part of the life process, yet it would be difficult to imagine millions of special-purpose chemical manufacturing units, one for each type of protein. Instead, a general-purpose mechanism assembles the proteins, guided by a description (think of it as a blueprint) that is contained in DNA (deoxyribonucleic acid) and RNA (ribonucleic acid) molecules. Both DNA and RNA are linear chains of small **nucleotides**; a DNA molecule might consist of more than a hundred million such nucleotides. In DNA there are four types of nucleotides, each consisting of some common structure and one of four different bases, named Adenine, Cytosine, Guanine, and Thymine. In RNA the structure is similar except that Thymine is replaced by Uracil.

The Genetic Code is a fixed-length description of how a sequence of nucleotides specifies an amino acid. Given that relationship, an entire protein can be specified by a linear sequence of nucleotides. Note that the coded description of a protein is not by itself any smaller or simpler than the protein itself; in fact, the number of atoms needed to specify a protein is larger than the number of atoms in the protein it represents. The benefit of the standardized representation is that it allows the same assembly apparatus to fabricate different proteins at different times.

Since there are four different nucleotides, one of them could specify at most four different amino acids. A sequence of two nucleotides could specify 16 different amino acids. But this is not enough—there are 20

different amino acids used in proteins—so a sequence of three is needed. Such a sequence is called a **codon**. There are 64 different codons, more than enough to specify 20 amino acids. The spare capacity is used to provide more than one combination for most amino acids, so the code is more robust. For example, Table 2.6 shows that the amino acid Alanine has 4 codes including all that start with GC; thus the third nucleotide can be ignored, so a mutation which changed it would not impair any biological functions. In fact, eight of the 20 amino acids have this same property that the third nucleotide is a "don't care." (It happens that the third nucleotide is more likely to be corrupted during transcription than the other two, due to an effect that has been called "wobble.")

An examination of the Genetic Code reveals that three codons (UAA, UAG, and UGA) do not specify any amino acid. These three signify the end of the protein. Such a "stop code" is necessary because different proteins are of different length. The codon AUG specifies the amino acid Methionine and also signifies the beginning of a protein; all protein chains begin with Methionine. Many man-made codes have this property, that most bit sequences designate data but a few are reserved for control information.

2.3.3 Telephone Area Codes

The third way in which spare capacity can be used is by reserving it for future expansion. The design issue is whether enough is reserved. When AT&T started using telephone Area Codes for the United States and Canada in 1947 (they were made available for public use in 1951), the codes contained three digits, with three restrictions.

- The first digit could not be 0 or 1, to avoid conflicts with 0 connecting to the operator, and 1 being an unintended effect of a faulty sticky rotary dial or a temporary circuit break of unknown cause (or today a signal that the person dialing acknowledges that the call may be a toll call)
- The middle digit could only be a 0 or 1 (0 for states and provinces with only one Area Code, and 1 for states and provinces with more than one). This restriction allowed an Area Code to be distinguished from an exchange (an exchange, the equipment that switched up to 10,000 telephone numbers, was denoted at that time by the first two letters of a word and one number; today exchanges are denoted by three digits).
- The last two digits could not be the same (numbers of the form *abb* are more easily remembered and therefore more valuable)—thus $x11$ dialing sequences such as 911 (emergency), 411 (directory assistance), and 611 (repair service) for local services were protected. This also permitted the later adoption of 500 (follow-me), 600 (Canadian wireless), 700 (interconnect services), 800 (toll-free calls), and 900 (added-value information services).

As a result only 144 Area Codes were possible. Initially 86 were used and were assigned so that numbers more rapidly dialed on rotary dials went to districts with larger incoming traffic (e.g., 212 for Manhattan). The remaining 58 codes were reserved for later assignment.

This pool of 58 new Area Codes was enough for more than four decades. Finally, when more than 144 Area Codes were needed, new Area Codes were created by relaxing the restriction that the middle digit be only 0 or 1. On January 15, 1995, the first Area Code with a middle digit other than 0 or 1 was put into service, in Alabama. The present restrictions on area codes are that the first digit cannot be 0 or 1, the middle digit cannot be 9, and the last two digits cannot be the same. As of the beginning of 2000, 108 new Area Codes had been started. This great demand was due in part to expanded use of the telephone networks for other services such as fax and cell phones, in part to political pressure from jurisdictions such as the Caribbean islands that wanted their own area codes, and in part by the large number of new telephone companies offering service and therefore needing at least one entire exchange in every rate billing district. Some people have suggested that the North American Numbering Plan (NANP) will run out of area codes before 2025, and there are various proposals for how to deal with that.

The transition in 1995 went remarkably smoothly, considering that every telephone exchange in North America required upgrading, both in revised software and, in some cases, new hardware. By and large the

public was not aware of the significance of the change. This was a result of the generally high quality of North American telephone service, and the fact that the industry was tightly coordinated. The only glitches seem to have been that a few PBX (Private Branch eXchanges) designed by independent suppliers were not upgraded in time. Since 1995 the telecommunications industry in North America has changed greatly: it now has less central control, much more competition, and a much wider variety of services offered. Any future changes in the numbering plan will surely result in much greater turmoil and inconvenience to the public.

2.3.4 IP Addresses

Another example of the need to reserve capacity for future use is afforded by IP (Internet Protocol) addresses, which is described in Section 2.8. These are (in version 4) of the form `x.x.x.x` where each `x` is an integer between 0 and 255, inclusive. Thus each Internet address can be coded in a total of 32 bits. IP addresses are assigned by the Internet Assigned Numbers Authority, http://www.iana.org/, (IANA).

The explosion of interest in the Internet has created a large demand for IP addresses, and the organizations that participated in the development of the Internet, who had been assigned large blocks of numbers, began to feel as though they were hoarding a valuable resource. Among these organizations are AT&T, BBN, IBM, Xerox, HP, DEC, Apple, MIT, Ford, Stanford, BNR, Prudential, duPont, Merck, the U.S. Postal Service, and several U.S. DoD agencies (see Section 2.8). The U.S. electric power industry, represented by EPRI (Electric Power Research Institute), requested a large number of Internet addresses, for every billable household or office suite, for eventual use by remote meter-reading equipment. The Internet Engineering Task Force, http://www.ietf.org/, (IETF) came to realize that Internet addresses were needed on a much more pervasive and finer scale than had been originally envisioned—for example, there will be a need for addresses for appliances such as refrigerators, ovens, telephones, and furnaces when these are Internet-enabled, and there will be several needed within every automobile and truck, perhaps one for each microprocessor and sensor on the vehicle. The result has been the development of version 6, IPv6, in which each address is still a sequence of four integers `x`, but each `x` is now a 32-bit number between 0 and 4,294,967,295. Thus new Internet addresses will require 128 bits. Existing addresses will not have to change, but all network equipment and protocols will have to change to accommodate the longer addresses. The new allocations include large blocks which are reserved for future expansion, and it is said (humorously) that there are blocks of addresses set aside for use by the other planets. The size of the address space is large enough to accommodate a unique hardware identifier for each personal computer, and some privacy advocates have pointed out that IPv6 may make anonymous Web surfing impossible.

2.3.5 ASCII

A fourth use for spare capacity in codes is to use some of it for denoting formatting or control operations. Many codes incorporate code patterns that are not data but control codes. For example, the Genetic Code includes three patterns of the 64 as stop codes to terminate the production of a protein.

The most commonly used code for text characters, ASCII (American Standard Code for Information Interchange, described in Section 2.5) reserves 33 of its 128 codes explicitly for control, and only 95 for characters. These 95 include the 26 upper-case and 26 lower-case letters of the English alphabet, the 10 digits, space, and 32 punctuation marks.

2.4 Extension of Codes

Many codes are designed by humans. Sometimes codes are amazingly robust, simple, easy to work with, and extendable. Sometimes they are fragile, arcane, complex, and defy even the simplest generalization.

Often a simple, practical code is developed for representing a small number of items, and its success draws attention and people start to use it outside its original context, to represent a larger class of objects, for purposes not originally envisioned. Codes that are generalized often carry with them unintended biases

from their original context. Sometimes the results are merely amusing, but in other cases such biases make the codes difficult to work with.

An example of a reasonably benign bias is the fact that ASCII has two characters that were originally intended to be ignored. ASCII started as the 7-bit pattern of holes on paper tape, used to transfer information to and from teletype machines. The tape originally had no holes (except a series of smaller holes, always present, to align and feed the tape), and travelled through a punch. The tape could be punched either from a received transmission, or by a human typing on a keyboard. The debris from this punching operation was known as "chad." The leader (the first part of the tape) was unpunched, and therefore represented, in effect, a series of the character 0000000 of undetermined length (0 is represented as no hole). Of course when the tape was read the leader should be ignored, so by convention the character 0000000 was called NUL and was ignored. Later, when ASCII was used in computers, different systems treated NULs differently. Unix treats NUL as the end of a word in some circumstances, and this use interferes with applications in which characters are given a numerical interpretation. The other ASCII code which was originally intended to be ignored is DEL, 1111111. This convention was helpful to typists who could "erase" an error by backing up the tape and punching out every hole. In modern contexts DEL is sometimes treated as a destructive backspace, but some text editors in the past have used DEL as a forward delete character, and sometimes it is simply ignored.

A much more serious bias carried by ASCII is the use of two characters, CR (carriage return) and LF (line feed), to move to a new printing line. The physical mechanism in teletype machines had separate hardware to move the paper (on a continuous roll) up, and reposition the printing element to the left margin. The engineers who designed the code that evolved into ASCII surely felt they were doing a good thing by allowing these operations to be performed separately. They could not have imagined the grief they have given to later generations as ASCII was adapted to situations with different hardware and no need to move the point of printing as called for by CR or LF separately. Different computing systems do things differently—Unix uses LF for a new line and ignores CR, Macintoshes (at least prior to OS X) use CR and ignore LF, and DOS/Windows requires both. This incompatibility is a continuing, serious source of frustration and errors. For example, in the transfer of files using FTP (File Transfer Protocol) CR and LF should be converted to suit the target platform for text files, but not for binary files. Some FTP programs infer the file type (text or binary) from the file extension (the part of the file name following the last period). Others look inside the file and count the number of "funny characters." Others rely on human input. These techniques usually work but not always. File extension conventions are not universally followed. Humans make errors. What should be done if part of a file is text and part binary?

2.5 Detail: ASCII

ASCII, which stands for "The American Standard Code for Information Interchange," was introduced by the American National Standards Institute (ANSI) in 1963. It is the most commonly used character code.

ASCII, a seven-bit code, represents the 33 control characters and 95 printing characters (including space) shown in Table 2.2. The control characters are used to signal special conditions, as described in Table 2.3.

Control Characters				Digits			Uppercase			Lowercase		
HEX	DEC	CHR	Ctrl	HEX	DEC	CHR	HEX	DEC	CHR	HEX	DEC	CHR
00	0	NUL	^@	20	32	SP	40	64	@	60	96	‘
01	1	SOH	^A	21	33	!	41	65	A	61	97	a
02	2	STX	^B	22	34	"	42	66	B	62	98	b
03	3	ETX	^C	23	35	#	43	67	C	63	99	c
04	4	EOT	^D	24	36	$	44	68	D	64	100	d
05	5	ENQ	^E	25	37	%	45	69	E	65	101	e
06	6	ACK	^F	26	38	&	46	70	F	66	102	f
07	7	BEL	^G	27	39	’	47	71	G	67	103	g
08	8	BS	^H	28	40	(	48	72	H	68	104	h
09	9	HT	^I	29	41	)	49	73	I	69	105	i
0A	10	LF	^J	2A	42	*	4A	74	J	6A	106	j
0B	11	VT	^K	2B	43	+	4B	75	K	6B	107	k
0C	12	FF	^L	2C	44	,	4C	76	L	6C	108	l
0D	13	CR	^M	2D	45	-	4D	77	M	6D	109	m
0E	14	SO	^N	2E	46	.	4E	78	N	6E	110	n
0F	15	SI	^O	2F	47	/	4F	79	O	6F	111	o
10	16	DLE	^P	30	48	0	50	80	P	70	112	p
11	17	DC1	^Q	31	49	1	51	81	Q	71	113	q
12	18	DC2	^R	32	50	2	52	82	R	72	114	r
13	19	DC3	^S	33	51	3	53	83	S	73	115	s
14	20	DC4	^T	34	52	4	54	84	T	74	116	t
15	21	NAK	^U	35	53	5	55	85	U	75	117	u
16	22	SYN	^V	36	54	6	56	86	V	76	118	v
17	23	ETB	^W	37	55	7	57	87	W	77	119	w
18	24	CAN	^X	38	56	8	58	88	X	78	120	x
19	25	EM	^Y	39	57	9	59	89	Y	79	121	y
1A	26	SUB	^Z	3A	58	:	5A	90	Z	7A	122	z
1B	27	ESC	^[	3B	59	;	5B	91	[	7B	123	{
1C	28	FS	^\	3C	60	¡	5C	92	\	7C	124	—
1D	29	GS	^]	3D	61	=	5D	93	]	7D	125	}
1E	30	RS	^^	3E	62	>	5E	94	ˆ	7E	126	˜
1F	31	US	^_	3F	63	?	5F	95	_	7F	127	DEL

Table 2.2: ASCII Character Set

On to 8 Bits

In an 8-bit context, ASCII characters follow a leading 0, and thus may be thought of as the "bottom half" of a larger code. The 128 characters represented by code points HEX 80–FF (sometimes incorrectly called "high ASCII" or "extended ASCII") have been defined differently in different contexts. On many operating systems they included the accented Western European letters and various additional punctuation marks. On IBM PCs they included line-drawing characters. Macs used (and still use) a different encoding. One

HEX	DEC	CHR	Ctrl	Meaning
00	0	NUL	^@	NULl blank leader on paper tape; generally ignored
01	1	SOH	^A	Start Of Heading
02	2	STX	^B	Start of TeXt
03	3	ETX	^C	End of TeXt; matches STX
04	4	EOT	^D	End Of Transmission
05	5	ENQ	^E	ENQuiry
06	6	ACK	^F	ACKnowledge; affirmative response to ENQ
07	7	BEL	^G	BELl; audible signal, a bell on early machines
08	8	BS	^H	BackSpace; nondestructive, ignored at left margin
09	9	HT	^I	Horizontal Tabulation
0A	10	LF	^J	Line Feed; paper up or print head down; new line on Unix
0B	11	VT	^K	Vertical Tabulation
0C	12	FF	^L	Form Feed; start new page
0D	13	CR	^M	Carriage Return; print head to left margin; new line on Macs
0E	14	SO	^N	Shift Out; start use of alternate character set
0F	15	SI	^O	Shift In; resume use of default character set
10	16	DLE	^P	Data Link Escape; changes meaning of next character
11	17	DC1	^Q	Device Control 1; if flow control used, XON, OK to send
12	18	DC2	^R	Device Control 2
13	19	DC3	^S	Device Control 3; if flow control used, XOFF, stop sending
14	20	DC4	^T	Device Control 4
15	21	NAK	^U	Negative AcKnowledge; response to ENQ
16	22	SYN	^V	SYNchronous idle
17	23	ETB	^W	End of Transmission Block
18	24	CAN	^X	CANcel; disregard previous block
19	25	EM	^Y	End of Medium
1A	26	SUB	^Z	SUBstitute
1B	27	ESC	^[	ESCape; changes meaning of next character
1C	28	FS	^\	File Separator; coarsest scale
1D	29	GS	^]	Group Separator; coarse scale
1E	30	RS	^^	Record Separator; fine scale
1F	31	US	^_	Unit Separator; finest scale
20	32	SP		SPace; usually not considered a control character
7F	127	DEL		DELete; originally ignored; sometimes destructive backspace

Table 2.3: ASCII control characters

widely used code is Windows CP-1252, which assigns useful characters in code points HEX 80–9F instead of little-used control characters.

There is, of course, no logical reason an operating system should choose how to interpret codes HEX 80–FF; that choice should be made by the application or the user. A Russian person might appreciate having the Cyrillic alphabet whereas scientists might want mathematical symbols. Many codes were defined, to support different languages or applications. Each country, each dialect, each special constituency, could choose its own code. The most widely known was ISO-8859-1 (ISO-Latin) which uses the 96 code points HEX A0–FF for various accented letters and punctuation of Western European languages.

Beyond 8 Bits

This approach of separate 128-character extensions is, of course, far too limited. What is a Russian mathematician to do? And how do you handle a Greek-Hebrew phrase book? And how about the Asian languages with their need for thousands of symbols? Fixed-length codes with 8 bits per symbol are simply inadequate.

This problem has a two-step solution. First, integer **code points** are assigned to letters and other symbols. Second, these code points are associated with bit arrays in a variable-length code.

The commonly accepted code-point assignment is **Unicode**, which associates an integer in the range 0 to 1,114,111 to letters and punctuation marks for all languages in use today (or at least that is the intention). The first 128 code points are the same as ASCII. The next 1,920 code points include letters and punctuation for languages with alphabets. The remaining code points cover the Asian languages and many other characters including mathematical symbols. Unicode was developed early, before any particular representation of these integers as bit patterns. The early development and acceptance of Unicode had the effect of inhibiting the design of many different codes for different special purposes.

In 1993 the very effective variable-length code **UTF-8** was designed to represent Unicode code points by sequences of 8-bit bytes. Characters in the range 0 through 127 (i.e., all ASCII characters) are represented by the ASCII bit patterns. Characters in the range 128 through 2,047 are represented as 2-byte (16-bit) sequences; characters 2,048 – 65,535 are represented in 3 bytes (24 bits), and the rest in 4 bytes. Thus UTF-8 is upward compatible from ASCII, is reasonably efficient for many languages, and is well suited for today's computers and communications networks. Although there have been other ways proposed to represent Unicode, UTF-8 is universally used today.

References

- One of many excellent ASCII charts, with PC and Windows 8-bit charts, and useful links, by Jim Price
  http://www.jimprice.com/jim-asc.shtml

- A Brief History of Character Codes, with a discussion of extension to Asian languages
  http://tronweb.super-nova.co.jp/characcodehist.html

- Unicode home page
  http://www.unicode.org/

- Windows CP-1252 standard
  http://www.microsoft.com/globaldev/reference/sbcs/1252.htm

- A few of many comparisons of ASCII, CP-1252, ISO-8859-1, Mac OS, Unicode, and HTML entities:
  - http://ftp.unicode.org/Public/MAPPINGS/VENDORS/MICSFT/WINDOWS/CP1252.TXT
  - http://www.alanwood.net/demos/ansi.html
  - http://www.jwz.org/doc/charsets.html

2.6 Detail: Integer Codes

There are many ways to represent integers as bit patterns. All suffer from an inability to represent arbitrarily large integers in a fixed number of bits. A computation which produces an out-of-range result is said to overflow.

The most commonly used representations are binary code for unsigned integers (e.g., memory addresses), 2's complement for signed integers (e.g., ordinary arithmetic), and binary gray code for instruments measuring changing quantities.

The following table gives five examples of integer codes. To keep the table small, only codes for 4 bits are shown, but each code can be defined for any number of bits. The **MSB** (most significant bit) is on the left and the **LSB** (least significant bit) on the right.

	Unsigned Integers		Signed Integers		
	Binary Code	Binary Gray Code	2's Complement	Sign/Magnitude	1's Complement
Range ⟶	[0, 15]	[0, 15]	[-8, 7]	[-7,7]	[-7,7]
-8			1 0 0 0		
-7			1 0 0 1	1 1 1 1	1 0 0 0
-6			1 0 1 0	1 1 1 0	1 0 0 1
-5			1 0 1 1	1 1 0 1	1 0 1 0
-4			1 1 0 0	1 1 0 0	1 0 1 1
-3			1 1 0 1	1 0 1 1	1 1 0 0
-2			1 1 1 0	1 0 1 0	1 1 0 1
-1			1 1 1 1	1 0 0 1	1 1 1 0
0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0
1	0 0 0 1	0 0 0 1	0 0 0 1	0 0 0 1	0 0 0 1
2	0 0 1 0	0 0 1 1	0 0 1 0	0 0 1 0	0 0 1 0
3	0 0 1 1	0 0 1 0	0 0 1 1	0 0 1 1	0 0 1 1
4	0 1 0 0	0 1 1 0	0 1 0 0	0 1 0 0	0 1 0 0
5	0 1 0 1	0 1 1 1	0 1 0 1	0 1 0 1	0 1 0 1
6	0 1 1 0	0 1 0 1	0 1 1 0	0 1 1 0	0 1 1 0
7	0 1 1 1	0 1 0 0	0 1 1 1	0 1 1 1	0 1 1 1
8	1 0 0 0	1 1 0 0			
9	1 0 0 1	1 1 0 1			
10	1 0 1 0	1 1 1 1			
11	1 0 1 1	1 1 1 0			
12	1 1 0 0	1 0 1 0			
13	1 1 0 1	1 0 1 1			
14	1 1 1 0	1 0 0 1			
15	1 1 1 1	1 0 0 0			

Table 2.4: Four-bit integer codes

Binary Code

This code is for nonnegative integers only. For code of length n, the 2^n patterns represent integers 0 through $2^n - 1$. The LSB (least significant bit) is 0 for even and 1 for odd integers.

Binary Gray Code

This code is for nonnegative integers. For code of length n, the 2^n patterns represent integers 0 through $2^n - 1$. The bit patterns of any two adjacent integers differ in exactly one bit. This property makes the code useful for sensors where the integer being encoded might change while a measurement is in progress. The following anonymous tribute appeared in Martin Gardner's column "Mathematical Games" in Scientific American, August, 1972, but actually was known much earlier.

The Binary Gray Code is fun,
for with it STRANGE THINGS can be done...
Fifteen, as you know,
is one oh oh oh,
while ten is one one one one.

2's Complement

This code is for integers, both positive and negative. For a code of length n, the 2^n patterns represent integers -2^{n-1} through $2^{n-1} - 1$. The LSB (least significant bit) is 0 for even and 1 for odd integers. Where they overlap, this code is the same as binary code. This code is widely used.

Sign/Magnitude

This code is for integers, both positive and negative. For code of length n, the 2^n patterns represent integers $-(2^{n-1} - 1)$ through $2^{n-1} - 1$. The MSB (most significant bit) is 0 for positive and 1 for negative integers; the other bits carry the magnitude. Where they overlap, this code is the same as binary code. While conceptually simple, this code is awkward in practice. Its separate representations for +0 and -0 are not generally useful.

1's Complement

This code is for integers, both positive and negative. For code of length n, the 2^n patterns represent integers $-(2^{n-1} - 1)$ through $2^{n-1} - 1$. The MSB is 0 for positive integers; negative integers are formed by complementing each bit of the corresponding positive integer. Where they overlap, this code is the same as binary code. This code is awkward and rarely used today. Its separate representations for +0 and -0 are not generally useful.

2.7 Detail: The Genetic Code*

The basic building block of your body is a cell. Two or more groups of cells form tissues, such as bone or muscle; tissues organize to form organs, such as the heart or brain; organs form organ systems, such as the circulatory system or nervous system; the organ systems together form you, the organism. Cells can be classified as either eukaryote or prokaryote cells—with or without a nucleus, respectively. The cells that make up your body and those of all animals, plants, and fungi are eukaryotic. Prokaryotes are bacteria and cyanobacteria.

The nucleus forms a separate compartment from the rest of the cell body; this compartment serves as the central storage center for all the hereditary information of the eukaryote cells. All of the genetic information that forms the book of life is stored on individual chromosomes found within the nucleus. In healthy humans there are 23 pairs of chromosomes (46 total). Each one of the chromosomes contains one threadlike deoxyribonucleic acid (DNA) molecule. Genes are the functional regions along these DNA strands, and are the fundamental physical units that carry hereditary information from one generation to the next. In the prokaryotes the chromosomes are free floating in the cell body since there is no nucleus.

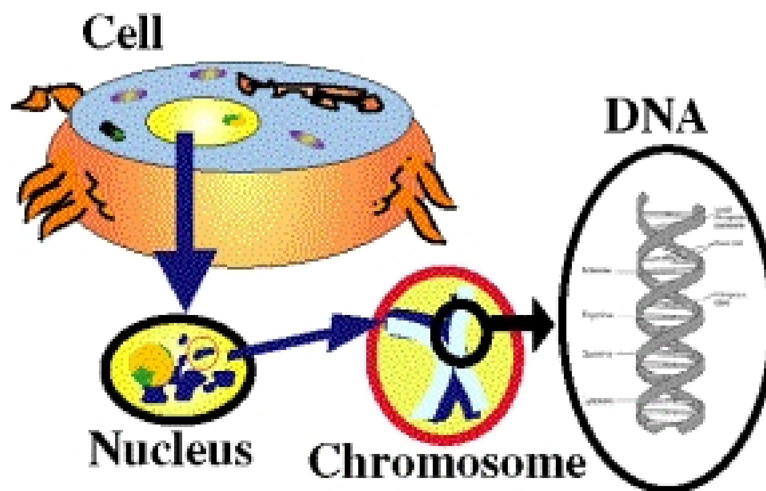

Figure 2.2: Location of DNA inside of a Cell

The DNA molecules are composed of two interconnected chains of nucleotides that form one DNA strand. Each nucleotide is composed of a sugar, phosphate, and one of four bases. The bases are adenine, guanine, cytosine, and thymine. For convenience each nucleotide is referenced by its base; instead of saying deoxyguanosine monophosphate we would simply say guanine (or G) when referring to the individual nucleotide. Thus we could write CCACCA to indicate a chain of interconnected cytosine-cytosine-adenine-cytosine-cytosine-adenine nucleotides.

The individual nucleotide chains are interconnected through the pairing of their nucleotide bases into a single double helix structure. The rules for pairing are that cytosine always pairs with guanine and thymine always pairs with adenine. These DNA chains are replicated during somatic cell division (that is, division of all cells except those destined to be sex cells) and the complete genetic information is passed on to the resulting cells.

Genes are part of the chromosomes and coded for on the DNA strands. Individual functional sections of the threadlike DNA are called genes. The information encoded in genes directs the maintenance and development of the cell and organism. This information travels a path from the input to the output: DNA (genes) $\Rightarrow$ mRNA (messenger ribonucleic acid) $\Rightarrow$ ribosome/tRNA $\Rightarrow$ Protein. In essence the protein is the final output that is generated from the genes, which serve as blueprints for the individual proteins.

*This section is based on notes written by Tim Wagner

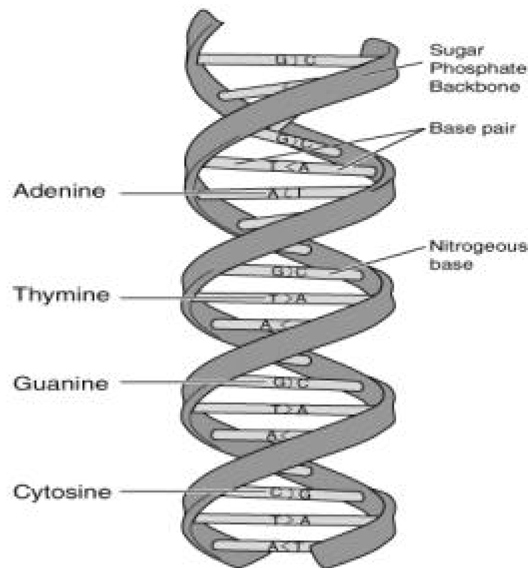

Figure 2.3: A schematic of DNA showing its helical structure

The proteins themselves can be structural components of your body (such as muscle fibers) or functional components (enzymes that help regulate thousands of biochemical processes in your body). Proteins are built from polypeptide chains, which are just strings of amino acids (a single polypeptide chain constitutes a protein, but often functional proteins are composed of multiple polypeptide chains).

The genetic message is communicated from the cell nucleus's DNA to ribosomes outside the nucleus via messenger RNA (ribosomes are cell components that help in the eventual construction of the final protein). Transcription is the process in which messenger RNA is generated from the DNA. The messenger RNA is a copy of a section of a single nucleotide chain. It is a single strand, exactly like DNA except for differences in the nucleotide sugar and that the base thymine is replaced by uracil. Messenger RNA forms by the same base pairing rule as DNA except T is replaced by U (C to G, U to A).

This messenger RNA is translated in the cell body, with the help of ribosomes and tRNA, into a string of amino acids (a protein). The ribosome holds the messenger RNA in place and the transfer RNA places the appropriate amino acid into the forming protein, illustrated schematically in Figure 2.4.

The messenger RNA is translated into a protein by first docking with a ribosome. An initiator tRNA binds to the ribosome at a point corresponding to a start codon on the mRNA strand—in humans this corresponds to the AUG codon. This tRNA molecule carries the appropriate amino acid called for by the codon and matches up at with the mRNA chain at another location along its nucleotide chain called an anticodon. The bonds form via the same base pairing rule for mRNA and DNA (there are some pairing exceptions that will be ignored for simplicity). Then a second tRNA molecule will dock on the ribosome of the neighboring location indicated by the next codon. It will also be carrying the corresponding amino acid that the codon calls for. Once both tRNA molecules are docked on the ribosome the amino acids that they are carrying bond together. The initial tRNA molecule will detach leaving behind its amino acid on a now growing chain of amino acids. Then the ribosome will shift over one location on the mRNA strand to make room for another tRNA molecule to dock with another amino acid. This process will continue until a stop codon is read on the mRNA; in humans the termination factors are UAG, UAA, and UGA. When the stop codon is read the chain of amino acids (protein) will be released om the ribosome structure.

What are amino acids? They are organic compounds with a central carbon atom, to which is attached by covalent bonds

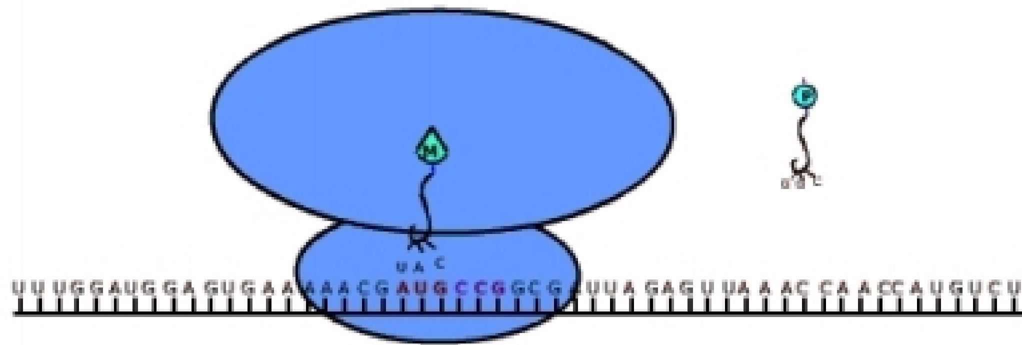

Figure 2.4: RNA to Protein transcription (click on figure for the online animation at http://www-mtl.mit.edu/Courses/6.050/2013/notes/rna-to-proteins.html)

- a single hydrogen atom H
- an amino group NH_2
- a carboxyl group COOH
- a side chain, different for each amino acid

The side chains range in complexity from a single hydrogen atom (for the amino acid glycine), to structures incorporating as many as 18 atoms (arginine). Thus each amino acid contains between 10 and 27 atoms. Exactly twenty different amino acids (sometimes called the "common amino acids") are used in the production of proteins as described above. Ten of these are considered "essential" because they are not manufactured in the human body and therefore must be acquired through eating (arginine is essential for infants and growing children). Nine amino acids are hydrophilic (water-soluble) and eight are hydrophobic (the other three are called "special"). Of the hydrophilic amino acids, two have net negative charge in their side chains and are therefore acidic, three have a net positive charge and are therefore basic; and four have uncharged side chains. Usually the side chains consist entirely of hydrogen, nitrogen, carbon, and oxygen atoms, although two (cysteine and methionine) have sulfur as well.

There are twenty different common amino acids that need to be coded and only four different bases. How is this done? As single entities the nucleotides (A,C,T, or G) could only code for four amino acids, obviously not enough. As pairs they could code for 16 (4^2) amino acids, again not enough. With triplets we could code for 64 (4^3) possible amino acids—this is the way it is actually done in the body, and the string of three nucleotides together is called a codon. Why is this done? How has evolution developed such an inefficient code with so much redundancy? There are multiple codons for a single amino acid for two main biological reasons: multiple tRNA species exist with different anticodons to bring certain amino acids to the ribosome, and errors/sloppy pairing can occur during translation (this is called wobble).

Codons, strings of three nucleotides, thus code for amino acids. In the tables below are the genetic code, from the messenger RNA codon to amino acid, and various properties of the amino acids[1] In the tables below * stands for (U, C, A, or G); thus CU* could be either CUU, CUC, CUA, or CUG.

[1] shown are the one-letter abbreviation for each, its molecular weight, and some of its properties, taken from H. Lodish, D. Baltimore, A. Berk, S. L. Zipursky, P. Matsudaira, and J. Darnell, "Molecular Cell Biology," third edition, W. H. Freeman and Company, New York, NY; 1995.

First Nucleotide Base of mRNA Codon	Second Nucleotide Base of mRNA Codon: U	C	A	G
U	UUU = Phe UUC = Phe UUA = Leu UUG = Leu	UC* = Ser	UAU = Tyr UAC = Tyr UAA = stop UAG = stop	UGU = Cys UGC = Cys UGA = stop UGG = Trp
C	CU* = Leu	CC* = Pro	CAU = His CAC = His CAA = Gln CAG = Gln	CG* = Arg
A	AUU = Ile AUC = Ile AUA = Ile AUG = Met (start)	AC* = Thr	AAU = Asn AAC = Asn AAA = Lys AAG = Lys	AGU = Ser AGC = Ser AGA = Arg AGG = Arg
G	GU* = Val	GC* = Ala	GAU = Asp GAC = Asp GAA = Glu GAG = Glu	GG* = Gly

Table 2.5: Condensed chart of Amino Acids. The asterisk * stands for any of the four nucleotides

Symbols		Amino Acid	M Wt	Properties		Codon(s)
Ala	A	Alanine	89.09	Non-essential	Hydrophobic	GC*
Arg	R	Arginine	174.20	Essential	Hydrophilic, basic	CG* AGA AGG
Asn	N	Asparagine	132.12	Non-essential	Hydrophilic, uncharged	AAU AAC
Asp	D	Aspartic Acid	133.10	Non-essential	Hydrophilic, acidic	GAU GAC
Cys	C	Cysteine	121.15	Non-essential	Special	UGU UGC
Gln	Q	Glutamine	146.15	Non-essential	Hydrophilic, uncharged	CAA CAG
Glu	E	Glutamic Acid	147.13	Non-essential	Hydrophilic, acidic	GAA GAG
Gly	G	Glycine	75.07	Non-essential	Special	GG*
His	H	Histidine	155.16	Essential	Hydrophilic, basic	CAU CAC
Ile	I	Isoleucine	131.17	Essential	Hydrophobic	AUU AUC AUA
Leu	L	Leucine	131.17	Essential	Hydrophobic	UUA UUG CU*
Lys	K	Lysine	146.19	Essential	Hydrophilic, basic	AAA AAG
Met	M	Methionine	149.21	Essential	Hydrophobic	AUG
Phe	F	Phenylalanine	165.19	Essential	Hydrophobic	UUU UUC
Pro	P	Proline	115.13	Non-essential	Special	CC*
Ser	S	Serine	105.09	Non-essential	Hydrophilic, uncharged	UC* AGU AGC
Thr	T	Threonine	119.12	Essential	Hydrophilic, uncharged	AC*
Trp	W	Tryptophan	204.23	Essential	Hydrophobic	UGG
Tyr	Y	Tyrosine	181.19	Non-essential	Hydrophobic	UAU UAC
Val	V	Valine	117.15	Essential	Hydrophobic	GU*
start		Methionine				AUG
stop						UAA UAG UGA

Table 2.6: The Amino Acids and some properties. The asterisk * stands for any of the four nucleotides

2.8 Detail: IP Addresses

Table 2.7 is taken from IPv4, http://www.iana.org/assignments/ipv4-address-space (version 4, which is in the process of being phased out in favor of version 6). IP addresses are assigned by the Internet Assigned Numbers Authority (IANA), http://www.iana.org/.

IANA is in charge of all "unique parameters" on the Internet, including IP (Internet Protocol) addresses. Each domain name is associated with a unique IP address, consisting of four integers between 0 and 255 separated by periods, e.g. 204.146.46.8, which systems use to direct information through the network.

Internet Protocol Address Space

The allocation of Internet Protocol version 4 (IPv4) address space to various registries is listed here. Originally, all the IPv4 address space was managed directly by the IANA. Later, parts of the address space were allocated to various other registries to manage for particular purposes or regions of the world. RFC 1466 documents most of these allocations.

Address Block	Registry - Purpose	Date
3.x.x.x	General Electric Company	May 1994
4.x.x.x	Bolt Beranek and Newman Inc.	Dec 1992
8.x.x.x	Bolt Beranek and Newman Inc.	Dec 1992
9.x.x.x	IBM	Aug 1992
11.x.x.x	DoD Intel Information Systems	May 1993
12.x.x.x	AT & T Bell Laboratories	Jun 1995
13.x.x.x	Xerox Corporation	Sep 1991
15.x.x.x	Hewlett-Packard Company	Jul 1994
16.x.x.x	Digital Equipment Corporation	Nov 1994
17.x.x.x	Apple Computer Inc.	Jul 1992
18.x.x.x	MIT	Jan 1994
19.x.x.x	Ford Motor Company	May 1995
20.x.x.x	Computer Sciences Corporation	Oct 1994
25.x.x.x	Royal Signals and Radar Establishment	Jan 1995
32.x.x.x	Norsk Informasjonsteknology	Jun 1996
34.x.x.x	Halliburton Company	Mar 1993
35.x.x.x	Merit Computer Network	Apr 1994
36.x.x.x	Stanford University	Apr 1993
38.x.x.x	PSINet, Inc.	Sep 1994
40.x.x.x	Eli Lilly and Company	Jun 1994
43.x.x.x	Japan Inet	Jan 1991
44.x.x.x	Amateur Radio Digital Communiccations	Jul 1992
46.x.x.x	Bolt Beranek and Newman Inc.	Dec 1992
47.x.x.x	Bell Northern Research	Jan 1991
48.x.x.x	Prudential Securities Inc.	May 1995
52.x.x.x	E. I. duPont de Nemours and Co., Inc.	Dec 1991
54.x.x.x	Merck and Co., Inc.	Mar 1992
56.x.x.x	U. S. Postal Service	Jun 1994
57.x.x.x	SITA	May 1995

Table 2.7: Original IP Address Assignments, partial list

2.9 Detail: Morse Code

Samuel F. B. Morse (1791–1872) was a landscape and portrait painter from Charlestown, MA. He frequently travelled from his home in New Haven, CT or his studio in New York City to work with clients across the nation. He was in Washington, DC in 1825 when his wife Lucretia died suddenly of heart failure. Morse learned of this event as rapidly as was possible at the time, through a letter sent to Washington, but it was too late for him to return in time for her funeral.

As a painter Morse met with only moderate success. Although his paintings can be found today in major museums—the Museum of Fine Arts, Boston, has seven—he never had an important impact on contemporary art. It was as an inventor that he is best known. (He combined his interest in technology and his passion for art in an interesting way: in 1839 he learned the French technique of making daguerreotypes and for a few years supported himself by teaching it to others.)

Returning from Europe in 1832, he happened to meet a fellow passenger who had visited the great European physics laboratories. He learned about the experiments of Ampère, Franklin, and others wherein electricity passed instantaneously over any known length of wire. Morse realized this meant that intelligence could be transmitted instantaneously by electricity. He understood from the circumstances of his wife's death the importance of rapid communication. Before his ship even arrived in New York he invented the first version of what is today called Morse Code. His later inventions included the hand key and some receiving devices. It was in 1844 that he sent his famous message WHAT HATH GOD WROUGHT from Washington to Baltimore. That event caught the public fancy, and produced national excitement not unlike the Internet euphoria 150 years later.

Morse Code consists of a sequence of short and long pulses or tones (dots and dashes) separated by short periods of silence. A person generates Morse Code by making and breaking an electrical connection on a hand key, and the person on the other end of the line listens to the sequence of dots and dashes and converts them to letters, spaces, and punctuation. The modern form of Morse Code is shown in Table 2.8. Note that messages are case-insensitive. The at sign was added in 2004 to accommodate email addresses. Two of the dozen or so control codes are shown. Non-English letters and some of the less used punctuation marks are omitted.

A	·—	K	—·—	U	··—	0	—————	Question mark	··——··
B	—···	L	·—··	V	···—	1	·————	Apostrophe	·————·
C	—·—·	M	——	W	·——	2	··———	Parenthesis	—·——·—
D	—··	N	—·	X	—··—	3	···——	Quotation mark	·—··—·
E	·	O	———	Y	—·——	4	····—	Fraction bar	—··—·
F	··—·	P	·——·	Z	——··	5	·····	Equals	—···—
G	——·	Q	——·—	Period	·—·—·—	6	—····	Slash	—··—·
H	····	R	·—·	Comma	——··——	7	——···	At sign	·——·—·
I	··	S	···	Hyphen	—····—	8	———··	Delete prior word	········
J	·———	T	—	Colon	———···	9	————·	End of Transmission	·—·—·

Table 2.8: Morse Code

If the duration of a dot is taken to be one unit of time then that of a dash is three units. The space between the dots and dashes within one character is one unit, that between characters is three units, and that between words seven units. Space is not considered a character, as it is in ASCII.

Unlike ASCII, Morse Code is a variable-length code. Morse realized that some letters in the English alphabet are more frequently used than others, and gave them shorter codes. Thus messages could be transmitted faster on average, than if all letters were equally long. Table 2.9 shows the frequency of the letters in written English (the number of times each letter is, on average, found per 1000 letters).

Morse Code was well designed for use on telegraphs, and it later saw use in radio communications before AM radios could carry voice. Until 1999 it was a required mode of communication for ocean-going vessels, even though by that time it was rarely used. Until 2007 ability to send and receive Morse Code was a

132	E	61	S	24	U
104	T	53	H	20	G, P, Y
82	A	38	D	19	W
80	O	34	L	14	B
71	N	29	F	9	V
68	R	27	C	4	K
63	I	25	M	1	X, J, Q, Z

Table 2.9: Relative frequency of letters in written English

requirement for U.S. citizens who wanted some types of amateur radio license, and it is still widely used by radio amateurs.

Since Morse Code is designed to be heard, not seen, Table 2.8 is only marginally useful. You cannot learn Morse Code from looking at the dots and dashes on paper; you have to hear them. If you want to listen to it on text of your choice, try a synthesizer on the Internet, such as

- http://morsecode.scphillips.com/jtranslator.html

A comparison of Tables 2.8 and 2.9 reveals that Morse did a good but not perfect job of assigning short sequences to the more common letters (E is the most common and the shortest, but the frequently used O is longer than it should be). It is said that he learned the letter frequencies not by counting letters in books and newspapers, but by visiting a print shop. The printing presses at the time used movable lead type, assembled by humans to form lines of type. Type was available for each letter in every font and size, typically stored in compartments in two wooden cases. Morse assumed that the printers knew their business and he simply counted how many of each letter they stocked. The two type cases were typically arranged one above the other, with the lesser used capital letters in the less accessible top case. Printers referred to those as “uppercase” letters.