# Codes and Communication,
# or
# *How to talk to the computer*

ISC1xxx

When we talk to a computer, or "give it input", we are sending it many kinds of information:

- numbers to add
- text to store
- mail messages to send
- music
- photographs
- movies

Somehow, the computer has to store all this information, and it has to do it using electronic hardware.

Computer information is stored in “memory”. We can think of computer memory as a huge array of boxes. Each box has a numeric label, and can hold a small amount of information.

One way to give the computer information is to create a file with a text editor. A file has a name, such as “apology.txt”, and you might have entered the following information:

```
Aunt Polly,
Sorry I was late!
Tom
```

Your information gets stored in a sequence of empty boxes:

```
Address  Contents
-------  --------
1000000  Aunt Pol
1000001  ly,@Sorr
1000002  y I was
1000003  late!@To
1000004  m&------
```

The @ represents the carriage return or **CR** key you pressed twice. The return isn't in the same class as alphabetic characters but it's just as important in formatting your message.

The & represents the "end of file" symbol or **EOF**.

The trailing -'s are unused memory that we didn't actually need for this message.

The computer "remembers" that "apology.txt" is stored starting at location 1000000, up to the EOF symbol.

We pause for a few remarks:

- In our example, the "apology.txt" file was stored in neighboring boxes, that is, 1000000, 1000001, and so on. Because computer memory is constantly being erased and reused, it may be necessary to store large files in a somewhat scattered fashion. The computer knows how to find all the scattered pieces in the right order whenever it needs them.
- What we have called "boxes" are technically called computer "words". A modern "64 bit" computer will use words that can hold 8 characters, as we saw in our example. Computer words are also called "memory locations" because every word has an address which enables the computer to find it.
- We will say that a computer file is an item which has a name, perhaps an associated icon, an address, and an end-of-file marker, containing a set of information of interest to a user. Most user activities will involve information stored as a file.

Let's continue with the idea of storing a file such as "apology.txt" on the computer. We saw that the first word of memory (at address 1000000) contained this fragment:

```
Aunt Pol
```

Since computer memory can actually only store 1's and 0's, we must be missing something here. It's convenient to think of the computer word as containing typewriter characters, but that's not what goes on. Actually, every character is stored in the computer using the ASCII code. This is simply a table that translates every character into a string of 8 0's and 1's. For instance the letter "A" has the ASCII code **01000001**.

Now we have room for 8 characters in the computer word. If we replace each character by an 8 digit string of 0's and 1's, we will get a string of 64 0's and 1's. Here's what happens to "Aunt Pol" after translation:

```
A        u        n        t        SPACE    P        o        l
01000001 01110101 01101110 01110100 00100000 01010000 01101111 01101100
```

The 8-binary digit ASCII Code for printable characters:

| | | | | | |
|---|---|---|---|---|---|
| space 00100000 | 0 00110000 | @ 01000000 | P 01010000 | ‘ 01100000 | p 01110000 |
| ! 00100001 | 1 00110001 | A 01000001 | Q 01010001 | a 01100001 | q 01110001 |
| " 00100010 | 2 00110010 | B 01000010 | R 01010010 | b 01100010 | r 01110010 |
| # 00100011 | 3 00110011 | C 01000011 | A 01010011 | c 01100011 | s 01110011 |
| $ 00100100 | 4 00110100 | D 01000100 | T 01010100 | d 01100100 | t 01110100 |
| % 00100101 | 5 00110101 | E 01000101 | U 01010101 | e 01100101 | u 01110101 |
| & 00100110 | 6 00110110 | F 01000110 | V 01010110 | f 01100110 | v 01110110 |
| ’ 00100111 | 7 00110111 | G 01000111 | W 01010111 | g 01100111 | w 01110111 |
| ( 00101000 | 8 00111000 | H 01001000 | X 01011000 | h 01101000 | x 01111000 |
| ) 00101001 | 9 00111001 | I 01001001 | Y 01011001 | i 01101001 | y 01111001 |
| * 00101010 | : 00111010 | J 01001010 | Z 01011010 | j 01101010 | z 01111010 |
| + 00101011 | ; 00111011 | K 01001011 | [ 01011011 | k 01101011 | { 01111011 |
| , 00101100 | < 00111100 | L 01001100 | \ 01011100 | l 01101100 | \| 01111100 |
| - 00101101 | = 00111101 | M 01001101 | ] 01011101 | m 01101101 | } 01111101 |
| . 00101110 | > 00111110 | N 01001110 | ^ 01011110 | n 01101110 | ~ 01111110 |
| / 00101111 | ? 00111111 | O 01001111 | _ 01011111 | o 01101111 | |

EOF 00000100 CR 00001111 HT 00001001

Note that EOF (end-of-file), CR (carriage return), and HT (horizontal tab) are usually considered “unprintable” characters, but they often occur in text files.

Now let's note four levels of information in the computer:

- bits: this is what we were calling 1's and 0's. A computer bit is the smallest unit of memory.
- bytes: this is hwat we were calling a letter or character; the term byte includes letters, digits, symbols, and more unusual things that don't have a printable version. A byte is made up of 8 bits. That means that there are 256 possible byte patterns
- words: we called these "boxes" before. Because each word in has an address, the computer can easily store and retrieve information in them. On typical computers, a word can hold up to 8 bytes.
- files: can be thought of as a sequence of computer words; Files are created and modified by the user, and can be almost any size.

If we plan to do numeric calculations, then the ASCII code is the wrong way to store our information. The IEEE format is a standard way of efficiently packing a wide range of numbers into the computer memory.

Let's start by thinking of the best way of planning to store integers, that is, whole numbers or counting numbers, into a single computer word, that is, a byte, the same amount of memory that can only store a single letter like 'W'.

A byte has 8 bits, so we can represent it as a string of 0's and 1's. Examples: 00000000, 00000001, 00110011, 10101010.

To write a particular byte, we have 2 choices for each bit. There are 8 choices to make, so there are 2*2*2*2*2*2*2*2=256 possible bytes. That suggests that there is enough room in a byte to represent any number from 0 to 255. (Computer people like to start counting at 0, not 1!)

We need a systematic rule to convert a number like 17 into a byte, and another rule to take any byte and turn it back into a number. The obvious idea is to think of the byte in terms of binary arithmetic.

Consider the string of digits 11010. Ordinarily, we would assume this represents a very big number, namely

1*10,000 + 1*1,000 + 0*100 + 1*10 + 0*1

But since the only digits we see are 1's and 0's, this might instead be a number written in binary arithmetic, in which case we interpret the number using multiples of 2:

1*16 + 1*8 + 0*4 + 1*2 + 0*1

which gives us a completely different result, namely 28 (going back to the decimal system).

Binary arithmetic is perfect for the computer, since the computer memory bits can be thought of as binary digits (which is where the word bits comes from..

A rule to turn a number **n** between 0 and 255 into a byte
**b7—b6—b5—b4—b3—b2—b1—b0**

```
replace n by n/2, and set the remainder to b0;
replace n by n/2, and set the remainder to b1;
replace n by n/2, and set the remainder to b2;
replace n by n/2, and set the remainder to b3;
replace n by n/2, and set the remainder to b4;
replace n by n/2, and set the remainder to b5;
replace n by n/2, and set the remainder to b6;
replace n by n/2, and set the remainder to b7;
```

Note that when we compute n/2, we always round the result down to a whole number.

Turn the number 135 into a byte

**b7—b6—b5—b4—b3—b2—b1—b0**

```
replace 135 by 135/2=67 with remainder 1.
replace  67 by  67/2=33 with remainder 1.
replace  33 by  33/2=16 with remainder 1.
replace  16 by  16/2= 8 with remainder 0.
replace   8 by   8/2= 4 with remainder 0.
replace   4 by   4/2= 2 with remainder 0.
replace   2 by   2/2= 1 with remainder 0.
replace   1 by   1/2= 0 with remainder 1.
```

This means the binary representation of 135 is 10000111.

Our integer-to-byte rule is pretty wordy. We could almost simplify it to two lines:

```
do this 8 times:
  replace n by n/2, and set the remainder to b?;
```

To make this work, we need to know that on the first step, we compute b0, and on the second step, b1, and so on. If we just keep track of our repetitions, we can handle this:

```
Do, for each number i from 0 to 7
  replace n by n/2, and set the remainder to b#i;
```

Now we need a rule to turn a byte **b7—b6—b5—b4—b3—b2—b1—b0** into a number **n** between 0 and 255:

```
set n to 0
replace n by n*2 and add b7;
replace n by n*2 and add b6;
replace n by n*2 and add b5;
replace n by n*2 and add b4;
replace n by n*2 and add b3;
replace n by n*2 and add b2;
replace n by n*2 and add b1;
replace n by n*2 and add b0;
```

We can check this with our previous result 10000111.

```
set n to 0
replace  0 by  0*2=  0 and add b7=1 to get   1;
replace  1 by  1*2=  2 and add b6=0 to get   2;
replace  2 by  2*2=  4 and add b5=0 to get   4;
replace  4 by  4*2=  8 and add b4=0 to get   8;
replace  8 by  8*2= 16 and add b3=0 to get  16;
replace 16 by 16*2= 32 and add b2=1 to get  33;
replace 33 by 33*2= 66 and add b1=1 to get  67;
replace 67 by 67*2=134 and add b0=1 to get 135;
```

Our byte-to-integer rule can also be simplified:

```
n = 0
Do, for each number i, counting down from 7 to 0
  replace n by 2*n+b#i
```

It makes some sense that, since in the integer-to-byte rule we divided, and count up from 0 to 7, that in the byte-to-integer rule we multiply, and count down from 7 to 0.

Our integer-to-byte and byte-to-integer rules are examples of algorithms, that is, definite rules for carrying out a computation.

Typically, an algorithm may have input quantities, that is, information to which the algorithm is applied, and output quantities, that is, information that is the result of the computation.

For the integer-to-byte rule, the input is the number $n$, and the output is the 8 bits $\{b7|b6|b5|b4|b3|b2|b1|b0\}$.

The other interesting thing about both algorithms is that they involve a step that is repeated several times, which suggests that we can simplify the description using some kind of repetition statement - sometimes called a do loop or for loop.