Systems

What is a computer?
WHAT IS A COMPUTER?

The term ‘computer’ originally referred to people whose job it was to perform repeated numerical calculations according to some pre-determined set of instructions, that is, an algorithm. At the beginning of modern computing, Alan Turing captured the essence of what human computers did, of what calculation or computation was: that all this could be understood as making or changing marks on paper according to some set of rules, and that those rules could be determined by the marks on the paper. This model became known as the Turing machine and it still forms one of the foundations of theoretical computer science.

Since the 1940s the term ‘computer’ has been used pretty much exclusively to refer to digital machines, which accept some sort of input data, process this according to some set of stored instructions, that is, a program, and output some sort of information. The power of digital computers comes from their ability to run through these stored instructions incredibly quickly: the chip at the heart of a modern smartphone might execute up to a couple of billion instructions per second! On the other hand, without programming, a computer can do nothing – it needs to be given instructions to follow.

You can think of digital technology as made up of two inter-related systems, the hardware which are the physical components, from processor and memory to power supply and screen, and the software which is the core operating system, embedded control programs, compilers or interpreters for high-level programming languages and all the many application programs used by or written by the computer’s user.

Computers now seem almost ubiquitous, with an incredible variety of electronic devices each having some sort of digital computer controlling how they operate, according to stored programs. It’s worth distinguishing between devices that contain computers, where the computer controls the operation of the device for one specific purpose, and more general programmable computers, where one computer can do many different things.

In the former category, the computer-controlled device, we might count digital watches, digital radios, digital televisions, computerised central heating controllers, digital cameras, the engine management system of a car or many, many other devices now commonplace. Even in these categories convergence of technologies and the internet of things\(^1\) has meant that previously ‘dumb’ digital devices such as watches, televisions, cameras and cars can now connect to the internet, have apps installed on them and, in some cases, be reprogrammed by their users.

When thinking about general-purpose computers (see Doctorow, 2012), it can sometimes be helpful to distinguish between those which the user themselves can program and those which can only run software written specifically for the device. For example, a smart TV or a games console could be thought of as a general-purpose computer, capable of doing many different things, and whilst it’s possible to create smart TV apps or write a video game, you would normally need to use another computer to do that. Originally, smartphones and tablet computers fell into this category too: they would only be able to run programs written and licenced for them on other systems, but tools such as TouchDevelop and Codea now mean that programs for devices like these can be written on the smartphone or tablet itself. Indeed, even some games consoles can be directly programmed, to a limited extent, using programs such as Kodu and Project Spark.

General purpose programmable computers, from laptops to the large and fast computers running data centres and ‘cloud computing’ are capable of running many, many different types of program, including the compilers or interpreters necessary to write and run programs in many different programming languages. At a theoretical level, if something can be computed by one system that meets certain basic conditions,\(^2\) it can be computed by any system that meets those conditions.

---

1. Physical objects that collect, share and access data via the internet; see, for example, https://en.wikipedia.org/wiki/Internet_of_Things
2. We call these conditions Turing completeness – systems that can simulate Turing’s theoretical computing machine can, in theory, simulate one another; see, for example, https://en.wikipedia.org/wiki/Turing_completeness
Whilst it’s not quite true that programming lets us solve any problem we could imagine, by using computational thinking processes to understand a problem and develop algorithms for solving it, and then to write the computer code which implements that algorithm as a program, on hardware that accepts input, produces output and connects to other machines, computers can be used to solve many, many interesting and difficult real-world problems, as well as allowing us to watch videos of cats playing the piano.

## Binary

All the data that computers work with, and all the instructions they follow, have to be represented as numbers. There are particular conventions or codes for particular types of data (or instructions). An understanding of the ways in which information can be represented, organised and processed by computers can be seen as of comparable importance to the concepts of computational thinking (Michaelson, 2015).

### Binary Numbers

Whereas we think of numbers as expressed in base 10, decimal, notation, expressing any (whole) number using our digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9, it’s much, much easier for computers to work with just two symbols, a 0 and 1, as each ‘switch’ in the computer can be simply set as off or on (more strictly, low or high voltages) to represent these – this is the case with modern integrated circuits as well as the relays, valves and discreet transistors that preceded them. Binary representation isn’t really fundamental to the ideas of computing (Brown, 2012), but the numerical representation of information and instructions is, and binary is important in the low level implementation of digital computing on current and past hardware.

The programme of study expect pupils to learn about binary:

- **understand how numbers can be represented in binary, and be able to carry out simple operations on binary numbers [for example, binary addition, and conversion between binary and decimal]**

3 For example, computers cannot solve the halting problem – that is, they cannot determine if any arbitrary code would terminate or not (Turing, 1936).

4 More strictly, low or high voltages.

---

In base 10 (decimal or denary notation), we use place value so that the same digit can represent different numbers:

<table>
<thead>
<tr>
<th>Thousands</th>
<th>Hundreds</th>
<th>Tens</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

Thus 2,395 is interpreted as two thousands, three hundreds, nine tens and five units, 2000+300+90+5. Note how each place is 10 times larger than the one that follows it.

A similar place value system works in binary, but the places carry twice the value of the following one:

<table>
<thead>
<tr>
<th></th>
<th>64</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

So 101010 is interpreted as one thirty-two, one eight and one two, 32+8+2, that is, 42 in base ten. Note that you **never** get more than one in each place in binary, so converting a binary number to a decimal one is simply the process of adding up the respective place values. Given the number 10011101, simply write out the place values of each bit (binary digit) starting at the right (the least significant bit), and doubling each time:

<table>
<thead>
<tr>
<th></th>
<th>128</th>
<th>64</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Then add up the values of the places where you have a 1: 128+16+8+4+1=157

This approach gives us an algorithm for converting from binary to decimal, which, of course we can implement as code (Figure 3.1):
Figure 3.1

```python
def bin2dec(binary='0'):
    decimal = 0
    place = 1
    for i in range(len(binary)-1,-1,-1):
        decimal = decimal + int(binary[i]) * place
        place = place * 2
    return decimal
```

Binary to decimal conversions in Snap! and Python. Python allows numbers to be input in binary, they are stored internally in binary, as all numbers are, but are printed in decimal. Thus `print(0b101010)` produces the output 42.

Going in the other direction is easy enough too. The easiest approach is to start with writing down place value headings, again starting at the right (the least significant bit) and doubling until you get to a column that would mean the next one would be bigger than the number. So with 150, we would have column headings:

128  64  32  16   8   4   2   1

We then start at the left, including any place we can, and keeping track of how much is left. Taking 150 as our example:

1   (and 22 left)
1 0 0 1   (and 6 left)
1 0 0 1 0   (and 2 left)
1 0 0 1 0 1   (and 0 left)
1 0 0 1 0 1 0

There are quicker approaches, for example we could use repeated division by two, keeping track of the remainders.

Figure 3.2

```python
def dec2bin(decimal=0):
    if decimal == 0:
        return '0'
    binary = ''
    while decimal > 0:
        binary = str(decimal % 2) + binary
        decimal = decimal // 2
    return binary
```

Decimal to binary conversion functions in Snap! and Python. Python has a built-in `bin` command to do this too.

Because of the repetition here, it’s relatively easy to code this algorithm (Figure 3.2):

150 / 2 = 75 r 0
75 / 2 = 37 r 1
37 / 2 = 18 r 1
18 / 2 = 9 r 0
9 / 2 = 4 r 1
4 / 2 = 2 r 0
2 / 2 = 1 r 0
1 / 2 = 0 r 1

The remainders then give the binary, in reverse order, so reading from the bottom up, we get 10010110 which is the binary representation for 150 as above.

It’s possible to represent numbers less than one in binary too, using the binary equivalent of decimal numbers, sometimes called bicimal. So just as in decimal we extend place value to the right as tenths, hundredths, thousandths, and so on, each place being a tenth the size of the previous one, so in binary we halve each time: one-half, one-quarter, one-eighth and so on.

Thus 3/8 would be 0.0111 in binary, as three-eighths = one-quarter + one-eighth. In binary recurring ‘bicimal’ numbers are quite common, so, for example, one-tenth would be represented as, that
is, 0.0001100110011… which will leave a rounding error wherever it’s truncated.

Given that computers only have limited memory, for very large or very small numbers it’s inefficient to store all the bits in a simple place value representation, so we use a floating point form, equivalent to scientific notation, storing both a mantissa and exponent. So just as \(1.14 \times 10^2\) is another representation of 114, so we could represent \(1,100,10\) as \(1.100,10 \times 2^{10}\) where the ‘bicimal’ (radix) point has been shifted six places (six being \(110\) in binary), storing the binary mantissa \((1.100,10)\) and exponent \((110)\) to represent decimal 114.

It’s also possible to store negative numbers in a binary representation. Given a fixed word length (the number of bits set aside for the number) the usual method is called ‘two’s compliment’: for a negative number we reverse the bits (0 becomes 1, 1 becomes 0) and add one. For example, 75 is \(100,111\) in binary:

\[
\begin{array}{cccccc}
128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
\end{array}
\]

but with two’s compliment and an eight-bit word (a byte), –75 would be \(101,101,001\), that is, \(101,101,01\).

\[
\begin{array}{cccccc}
128 & 64 & 32 & 16 & 8 & 4 & 2 & 1 \\
1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
\end{array}
\]

Note though that the first bit no longer represents 128s; it now shows 0 for positive and 1 for negative, thus instead of using eight bits to store numbers from 0 to 255 we instead would store –128 to 127.

It’s worth bearing in mind that binary and decimal are just different ways of representing the same number: forty-two is still forty-two, whether we write it as ‘forty-two’, \(42\), XLII or in binary as \(101010\). Don’t let pupils confuse the thing itself with the way the thing is represented.

### Arithmetic

Counting in binary is easy, and a really nice pattern quickly emerges. Start with one, changing the rightmost bit by one each time, carrying into the next column when you would get to two:

\[
\begin{array}{cccccc}
1 & 10 & 11 & 100 & 101 & 110 & 111 & 1000 & 1001 & 1010 & 1011 & 1100 & 1101 & 1110 & 1111 & 10000 & ... \\
\end{array}
\]

Like counting binary arithmetic is surprisingly easy to master, and a good way to revisit the standard algorithms for the four rules of arithmetic that pupils will have learnt in primary school. The key to this is to remember that we carry when we get to two, not 10, as we only ever have 1s and 0s in any place.

Look at the addition example in Figure 3.3, starting from the units column on the right. 0 plus 1 is 1, in the twos, 1 plus 0 is 1. In the fours, 1 plus 1 is two, so we put 0 down and carry the 1. The in the eights we have 0 plus 0 plus 1 which is 1 and so on. The sum and carry operations in binary addition here can be carried out using a relatively simple combination of logic gates, see page 99.

Subtraction can be done easily by hand too, in the example below (Figure 3.4) using the decomposition method that most schools use for decimal subtraction.
Again, starting from the right: 1 minus 1 is 0. In the twos column, 1 minus 0 is 1. In the fours we have a problem, as we can’t do 0 minus 1, so we decompose the eight into two (10) fours, then 10 minus 1 is 1 in binary. In the eights we have 0 minus 0, which is 0. In the sixteens we have a problem again, as we can’t do 0 minus 1. We can’t decompose the thirty-twos, so we decompose one sixty-four into two thirty-twos, then decompose one of them into two sixteens, and 10 minus 1 is 1 as before. Finally in the thirty-twos, 10 minus 1 is 1.

The times tables for binary are quite easy to learn:

- \(0 \times 0 = 0\)
- \(0 \times 1 = 0\)
- \(1 \times 0 = 0\)
- \(1 \times 1 = 1\)

which is the same as the truth table for Boolean AND, if we represent True as 1 and False as 0. Again, we can apply the usual decimal long multiplication to multiplication in binary (Figure 3.5):

Again, starting from the right of the number we’re multiplying by, in the units column, 1101 \(\times\) 1 is 1101; in the twos column, 1101 \(\times\) 0 is 0; in the fours column 1101 \(\times\) 1 is 1101, but we shift across into the four column (that is, a couple of places) to write this down, 1101 \(\times\) 100 = 110100. Adding these answers up using binary addition, we get 1000001. Notice that multiplication is simply repeated shifts of the original number together with binary addition, both of which are easy to accomplish in digital circuits.

The process here matches the ‘Egyptian’ or ‘Russian Peasant’ method for long multiplication:\(^5\) start by writing down the two numbers to be multiplied, the larger on the left, the smaller on the right. Double the numbers on the right, halve the numbers on the left, discarding any remainders:

\[
\begin{align*}
13 & \quad 5 \\
26 & \quad 2 \\
52 & \quad 1
\end{align*}
\]

Now, discard any lines with an even number on the right:

\[
\begin{align*}
13 & \quad 5 \\
52 & \quad 1
\end{align*}
\]

and then just add the numbers on the left:

\[13 + 52 = 65\]

The first step here, doubling the numbers on the left, is simply a left shift in binary. Halving and ignoring the even values is the equivalent of binary conversion, so we only add up the shifted values corresponding to a 1 in the binary representation.

Pupils can also do division in binary, again using the same algorithm as for decimal long division,\(^6\) but here with the advantage that the divisor either divides or does not divide into the dividend at each stage. There’s an argument that binary is a far better base for learning the mechanics of this algorithm, as the additional cognitive load of estimating how many times the divisor goes into the dividend at each step is removed.

---

5 See, for example, [www.cut-the-knot.org/Curriculum/Algebra/PeasantMultiplication.shtml](http://www.cut-the-knot.org/Curriculum/Algebra/PeasantMultiplication.shtml)

From Victorian times, way before the advent of digital computers and the internet, electrical circuits were used to transmit information, originally using simple, binary state, on-off switches (the telegraph), before the use of analogue sound signals (the telephone). Rather than converting text to numbers, a different form of representation was soon agreed on, in which each letter of the alphabet (plus punctuation, digits and other symbols) had an agreed sequence of short or long pulses associated with it — this was Morse code, named after its inventor (Figure 3.7).

![Figure 3.6](image1)

Working this time from the most significant bit of the dividend, on the left (Figure 3.6):

101 doesn’t go into 1. It doesn’t go into 10. It doesn’t go into 100. It does go into 1,000, 1 time, so we now work out the remainder in the eights column, using binary subtraction, 1000–101 = 11.

Bringing down the next bit, a 0, 101 does go into 110, 1 time, with a remainder of 110–101, that is, 1.

Bringing down the next bit, 0, 101 doesn’t go into 10, so we write 0 in the twos column of the quotient. Bringing down the next bit, 1, 101 goes into 101 1 time, with a remainder of 101–101, that is, 0.

![Figure 3.7](image2)

The key thing here is not the details of the representation, but the idea that there needed to be a single, agreed system for communicating via the telegraph’s infrastructure: the same held true with the adoption of digital computers, and, particularly, their connection via the internet. Morse’s system took account of the relative frequency of letters in English, thus e and t, the most frequently occurring letters in typical English prose, have very short symbols, . and - respectively, but other letters, such as q and j, which occur far less frequently, have longer pulse patterns, and thus take longer to transmit.

Another system, Baudot code,⁷ represented each letter as a pattern of five on or off signals, which subsequently became the standard for telegraph communications and the teleprinters used as terminals to communicate with the first computers. Five bits allowed only 32 different symbols to be represented, but shift codes were used to swap between letters.

---


**Text**

As well as being able to work with numbers in a binary form, the programme of study also expects pupils to understand how other forms of information are represented in a computer numerically:

understand how data of various types (including text, sounds and pictures) can be represented and manipulated digitally, in the form of binary digits

In order for computers to store, process or transmit information as text, it’s necessary for this to be coded as numbers (with the numbers themselves stored as binary).
and symbols (including numbers). A version of this, the International Telegraph Alphabet (ITA2) remains in use for some applications even today.

For a long time, the most widely adopted code was US-ASCII (United States-American Standard Code for Information Interchange; American Standards Association, 1963). In US-ASCII numbers from 0 to 127 are used to represent upper and lower case letters of the Latin alphabet, the digits 0–9, commonly used punctuation, and necessary control characters (such as new lines and backspace). By this point eight bits (enough for the numbers 0–255) had become a standard unit of memory, the byte, and one byte was thus more than enough to store or transmit any single character of standard English text, with room to spare if needed.

<table>
<thead>
<tr>
<th>Number</th>
<th>Binary</th>
<th>Character</th>
<th>Number</th>
<th>Binary</th>
<th>Character</th>
<th>Number</th>
<th>Binary</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>00100000</td>
<td>(space)</td>
<td>64</td>
<td>01000000</td>
<td>@</td>
<td>96</td>
<td>01100000</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>00100001</td>
<td>!</td>
<td>65</td>
<td>01000001</td>
<td>A</td>
<td>97</td>
<td>01100001</td>
<td>a</td>
</tr>
<tr>
<td>34</td>
<td>00100010</td>
<td>&quot;</td>
<td>66</td>
<td>01000010</td>
<td>B</td>
<td>98</td>
<td>01100010</td>
<td>b</td>
</tr>
<tr>
<td>35</td>
<td>00100011</td>
<td>#</td>
<td>67</td>
<td>01000011</td>
<td>C</td>
<td>99</td>
<td>01100011</td>
<td>c</td>
</tr>
<tr>
<td>36</td>
<td>00100100</td>
<td>$</td>
<td>68</td>
<td>01000100</td>
<td>D</td>
<td>100</td>
<td>01100100</td>
<td>d</td>
</tr>
<tr>
<td>37</td>
<td>00100101</td>
<td>%</td>
<td>69</td>
<td>01000101</td>
<td>E</td>
<td>101</td>
<td>01100101</td>
<td>e</td>
</tr>
<tr>
<td>38</td>
<td>00100110</td>
<td>&amp;</td>
<td>70</td>
<td>01000110</td>
<td>F</td>
<td>102</td>
<td>01100110</td>
<td>f</td>
</tr>
<tr>
<td>39</td>
<td>00100111</td>
<td>'</td>
<td>71</td>
<td>01000111</td>
<td>G</td>
<td>103</td>
<td>01100111</td>
<td>g</td>
</tr>
<tr>
<td>40</td>
<td>00101000</td>
<td>(</td>
<td>72</td>
<td>01001000</td>
<td>H</td>
<td>104</td>
<td>01101000</td>
<td>h</td>
</tr>
<tr>
<td>41</td>
<td>00101001</td>
<td>)</td>
<td>73</td>
<td>01001001</td>
<td>I</td>
<td>105</td>
<td>01101001</td>
<td>i</td>
</tr>
<tr>
<td>42</td>
<td>00101010</td>
<td>*</td>
<td>74</td>
<td>01001010</td>
<td>J</td>
<td>106</td>
<td>01101010</td>
<td>j</td>
</tr>
<tr>
<td>43</td>
<td>00101011</td>
<td>+</td>
<td>75</td>
<td>01001011</td>
<td>K</td>
<td>107</td>
<td>01101011</td>
<td>k</td>
</tr>
<tr>
<td>44</td>
<td>00101100</td>
<td>,</td>
<td>76</td>
<td>01001100</td>
<td>L</td>
<td>108</td>
<td>01101100</td>
<td>l</td>
</tr>
<tr>
<td>45</td>
<td>00101101</td>
<td>-</td>
<td>77</td>
<td>01001101</td>
<td>M</td>
<td>109</td>
<td>01101101</td>
<td>m</td>
</tr>
<tr>
<td>46</td>
<td>00101110</td>
<td>.</td>
<td>78</td>
<td>01001110</td>
<td>N</td>
<td>110</td>
<td>01101110</td>
<td>n</td>
</tr>
<tr>
<td>47</td>
<td>00101111</td>
<td>/</td>
<td>79</td>
<td>01001111</td>
<td>O</td>
<td>111</td>
<td>01101111</td>
<td>o</td>
</tr>
<tr>
<td>48</td>
<td>00110000</td>
<td>0</td>
<td>80</td>
<td>01010000</td>
<td>P</td>
<td>112</td>
<td>01110000</td>
<td>p</td>
</tr>
<tr>
<td>49</td>
<td>00110001</td>
<td>1</td>
<td>81</td>
<td>01010001</td>
<td>Q</td>
<td>113</td>
<td>01110001</td>
<td>q</td>
</tr>
<tr>
<td>50</td>
<td>00110010</td>
<td>2</td>
<td>82</td>
<td>01010010</td>
<td>R</td>
<td>114</td>
<td>01110010</td>
<td>r</td>
</tr>
<tr>
<td>51</td>
<td>00110011</td>
<td>3</td>
<td>83</td>
<td>01010011</td>
<td>S</td>
<td>115</td>
<td>01110011</td>
<td>s</td>
</tr>
<tr>
<td>52</td>
<td>00110100</td>
<td>4</td>
<td>84</td>
<td>01010100</td>
<td>T</td>
<td>116</td>
<td>01110100</td>
<td>t</td>
</tr>
<tr>
<td>53</td>
<td>00110101</td>
<td>5</td>
<td>85</td>
<td>01010101</td>
<td>U</td>
<td>117</td>
<td>01110101</td>
<td>u</td>
</tr>
<tr>
<td>54</td>
<td>00110110</td>
<td>6</td>
<td>86</td>
<td>01010110</td>
<td>V</td>
<td>118</td>
<td>01110110</td>
<td>v</td>
</tr>
<tr>
<td>55</td>
<td>00110111</td>
<td>7</td>
<td>87</td>
<td>01010111</td>
<td>W</td>
<td>119</td>
<td>01110111</td>
<td>w</td>
</tr>
<tr>
<td>56</td>
<td>00111000</td>
<td>8</td>
<td>88</td>
<td>01011000</td>
<td>X</td>
<td>120</td>
<td>01111000</td>
<td>x</td>
</tr>
<tr>
<td>57</td>
<td>00111001</td>
<td>9</td>
<td>89</td>
<td>01011001</td>
<td>Y</td>
<td>121</td>
<td>01111001</td>
<td>y</td>
</tr>
<tr>
<td>58</td>
<td>00111010</td>
<td>:</td>
<td>90</td>
<td>01011010</td>
<td>Z</td>
<td>122</td>
<td>01111010</td>
<td>z</td>
</tr>
<tr>
<td>59</td>
<td>00111100</td>
<td>;</td>
<td>91</td>
<td>01011100</td>
<td>[</td>
<td>123</td>
<td>01111100</td>
<td>{</td>
</tr>
<tr>
<td>60</td>
<td>00111101</td>
<td>&lt;</td>
<td>92</td>
<td>01011101</td>
<td>\</td>
<td>124</td>
<td>01111100</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>00111110</td>
<td>=</td>
<td>93</td>
<td>01011110</td>
<td>]</td>
<td>125</td>
<td>01111101</td>
<td>}</td>
</tr>
<tr>
<td>62</td>
<td>00111111</td>
<td>&gt;</td>
<td>94</td>
<td>01011111</td>
<td>^</td>
<td>126</td>
<td>01111110</td>
<td>~</td>
</tr>
<tr>
<td>63</td>
<td>00111111</td>
<td>?</td>
<td>95</td>
<td>01011111</td>
<td>_</td>
<td>127</td>
<td>01111111</td>
<td>(delete)</td>
</tr>
</tbody>
</table>

The spare capacity in US-ASCII (it only took seven bits of an eight bit byte) allowed other alphabets to be represented by the numbers 128–255, swapping in and out different code pages depending on the particular alphabet and language to be represented – thus Russian Cyrillic characters could be represented using the same numbers as would be used to represent Arabic characters, depending on the particular code page added on for the 128–255 range above standard US-ASCII.

Back in the days when memory was expensive and scarce, such a swappable code page system would work well enough if working with single, alphabet-based languages, but what hope would it have of representing the characters of a language such as Chinese?

Table 3.1
Subsequent development saw the extension or perhaps, more accurately, the replacement of ASCII with Unicode, which uses (up to) 32 bits, that is, four bytes, to store each character, representing linguistic (and other) symbols with numbers between 0 and 4,294,967,295, although thus far only 120,000 or so characters from 129 scripts are encoded. The characters coded with 0–127 in Unicode (UTF-8) match exactly the characters given these numbers in US-ASCII, so for those working with standard English there are few practical differences between the two systems. It’s fascinating to explore the full Unicode table to see the diversity of symbols used to write human languages.

A couple of lines of code allow text to be converted to its numeric representation and vice-versa.

```python
list(map(chr,[72,101,108,108,111]))
list(map(ord,"Hello"))
```

**Figure 3.8**

Snap! (Figure 3.8) and Python code to convert between character codes and text. Note that Snap! works in Unicode by default, whereas Python defaults to ASCII.

Editing text is, in essence, simply about making changes to the sequence of numbers which represent any particular string of characters. Familiar operations such as cut, copy and paste involve manipulating sequences of numbers: thus cut involves removing some numbers from the sequence, making a copy in another memory location (essentially a variable), copy involves duplicating part of the sequence, paste would be inserting one sequence within another. Whilst strings of characters and lists are typically thought of as different data structures, their internal representation as sequences of numbers will have much in common: and cut, copy and paste for text directly parallel common operations on lists.

One interesting development has been the way in which the US-ASCII punctuation and other characters have been combined in ways previously absent from language to convey succinctly emotions which would otherwise be cumbersome to convey in writing: thus emoticons such as the following are commonly used in online communication.

- :-)
- :-(
- ;-)
- 8-()

More recently, this is extended into the novel linguistic form of the emoji, in which pictorial representations of words can take the place of more conventional, character-based forms. Emojis too are represented as numbers, with many now being included in the Unicode table. For example Unicode 1F600 to 1F607 are the characters

![Emoji Characters](8)

The way in which a particular character is shown on screen or when printed out is different from its internal representation. Part of the job of the operating system and application software is to take the internal numerical representation of the character and display or print it as a specific glyph using a particular font, converting the code for the character into patterns in pixels, lines and curves, or ink that can be read on screen or paper respectively. Similarly, text-to-speech interfaces must take the character-by-character numerical representation, process this according to the grapheme/phoneme correspondence for the language and produce appropriate audio using one of perhaps several ‘voices’.

**Images**

There are two main ways to represent images digitally: the most common involves imposing some form of grid on the image, and then allocating numbers to the colour of each cell (square) in the grid – we call this a bitmap representation; an alternative is to describe the shapes (lines, curves, polygons, and so on) from which the image is made, essentially writing a program to reproduce the image from its components.

A colour bitmap then is made up of a (typically)

---

9 [http://unicode.org/emoji/charts/full-emoji-list.html](http://unicode.org/emoji/charts/full-emoji-list.html)
rectangular grid of (typically) small squares, called pixels, each of which is thought of as having one of a fixed number of possible values for red, green and blue components. Digital cameras take this approach for input, using a lens to focus light onto an array of light sensitive receptors, usually with red, green and blue filters in front. LCD (and similar) screens take the same approach to output, shining light through a semitransparent grid through which brightness can be controlled to a particular level, again with red, green, blue filters in front.

In storing bitmaps, as with any digitisation process, there’s a trade-off here between the amount of storage (memory) needed for the image and the resolution and colour fidelity stored.

Using just one bit per pixel reduces the representation to simply black or white, but even with only a few pixels it’s often possible to recognise the image (Figure 3.10–3.11):

Allowing 8 bits (one byte) per pixel allows 256 shades of grey to be represented, greatly improving the quality of the representation, but taking eight times as much memory as a black and white image
If we use three times the memory, using one byte (8 bits, 256 levels) for each red, green and blue colour channel, then we have the full 16 million colour representation that we are used to in digital media (Figure 3.14–3.15).

Figure 3.12 3,550 pixel bitmap, 8 bits per pixel, so 3.55kb of memory

Figure 3.13 120,000 pixel bitmap, 8 bits per pixel, so 120kb of memory

Figure 3.14 3,550 pixel image, 24 bits per pixel, so 10.65 kB of memory

Figure 3.15 120,000 pixel image, 24 bits per pixel, so 360kb of memory
Image manipulation software allows pupils to experiment with the effect of reducing the resolution of an image and the number of bits used to store the colour or brightness information. Pupils can also explore creating ‘pixel’ art, choosing the colour for each pixel of the image, typically at a very low resolution. They can do this by hand on gridded paper or using a spreadsheet, perhaps using conditional formatting tools to shade cells according to the number entered.

Mathematician and comedian Matt Parker makes an online tool available to convert images in standard file formats into suitably coloured-in Excel spreadsheets. Once the pixel colour values of the image are in a spreadsheet format, it’s easy enough to apply formulae to cells and groups of cells, to see how simple image manipulation can be accomplished: increasing the brightness of the image means increasing the values in each cell; reducing an image to greyscale involves replacing each colour value with the average of the red, green and blue values for a pixel; blurring an image can be accomplished by replacing each red, green or blue pixel value with the average of the corresponding values for the nine or 25 surrounding cells, and so on.

Similar effects, and indeed much more, can be accomplished in programming languages, for example using Python’s pillow or scikit-image libraries, or using the tools built in to standard image manipulation packages.

The other approach to working with images, vector graphics, in which we give the instructions for the lines and curves that make up an image has a number of advantages: the files here tend to be more compact, there’s no fundamental limit to the resolution at which images can be displayed, and there’s no ‘pixelation’. However, this approach is much more suitable for working with drawings created originally on the computer than for digitising images of the real world.

Audio

In the case of sound, again there are a couple of options: the first involves storing a sequence of numbers to represent the volume of sound at different points in time, the other would be closer to composing music, creating a set of instructions for sound to be played.

Let’s take the case of recording some sound on a computer. A microphone takes the pressure waves in the air that we hear and converts these into an analogue electrical signal. The analogue signal is then sampled lots of times a second and each of the sampled voltage values is then simply converted to a number. This is called pulse code modulation (PCM) and is used in the .WAV file format. ‘CD quality’ audio is sampled 44,100 times a second (that is, a sampling frequency of 44.1khz), and 16 bits are used to store the different sound or voltage values (that is, from 0 to 65,535), for both left and right channels. Thus one minute of CD quality, stereo audio takes just over 10 mb of storage.

As with image representation, there’s a trade-off between the storage capacity needed and the veracity of the digital representation. It’s possible to store reasonable audio with a lower quality than this, and for spoken-word recording this may suffice – mono recordings at 11khz storing just 8 bits for each sample are usually acceptable, with quality comparable to long wave radio or analogue telephone calls. It’s also possible to store at a higher quality – so called ‘high definition’ audio uses 24 bits for each sample, allowing 8,388,608 different audio intensity values to be stored, and sample rates as high as 192khz. Even here though, a digital representation can never be a perfect match to the even finer-grained analogue signal.

10 For example, Photoshop, The Gimp or Pixlr.
11 www.think-maths.co.uk/spreadsheet, qv www.youtube.com/watch?v=UBX2QOHiQ_1
12 Note: 2 channels × 2 bytes per sample × 44,100 samples a second × 60 seconds in a minute.
13 Note that 32bit, 384khz audio is also available.
Editing digital audio essentially means manipulating the sequence of numbers that represent the audio signal – making a recording louder could be accomplished by multiplying all the numbers by a number bigger than one; making the recording quieter involves multiplying by a number less than one; silencing part of a recording would mean replacing the audio signal values with 0; cutting a section of a recording involves deleting the numbers corresponding to that section from the sequence of numbers in the file.

It is possible to view and edit the contents of an audio file (for example the .WAV format) using a hexadecimal editor for binary files, although given the very high number of samples per second it’s hard work to do so in any meaningful way. With some ingenuity, .WAV audio files can be imported and exported from Excel or other spreadsheets, and the data manipulated directly using cell-based formulae. Audio files can also be manipulated as data in Python or other programming languages. Audio editing software such as Audacity provides a graphical user interface and simple, intuitive tools for working with audio files at a higher level of abstraction, hiding the numerical representation of the audio from the user.

![Original sound wave](image1)
![Analogue sound wave](image2)
![Digital sound wave](image3)

**Figure 3.16**

For music, it’s possible to think in terms of representing the composition digitally, rather than the sound that’s heard, essentially writing a sequence of instructions (a program) which when executed would play the music – the MIDI file format does this, storing the order of note values and durations that make up the music, and then using software to play these back, typically using short samples of audio from recorded instruments or digitally-generated (synthesized) tones.

It’s possible to create midi format files using a wide range of applications, including sequencing and traditional stave notation composition software, and pupils can get a feel for this more programmatic approach to music using Scratch or Sonic Pi, both of which use MIDI note values as a starting point.

```python
import wave
import random
import struct

noise_output = wave.open('noise.wav', 'w')
noise_output.setparams((2, 2, 44100, 0, 'NONE', 'not compressed'))

for i in range(0, 44100):
    value = random.randint(-32767, 32767)
    packed_value = struct.pack('h', value)
    noise_output.writeframes(packed_value)

noise_output.close()
```

**Python program to generate 1 second of stereo random noise in 16 bit 44.1 khz PCM format**

Often it is useful to take large text, image or sound files and store them in a more compact form, using fewer bytes to store the same, or almost the same, information. Although computer storage capacities have increased exponentially for unit cost (Walter, 2005), ever increasing amounts of data at higher and higher resolutions are stored. Moreover, more slowly increasing internet bandwidth is used to transmit these files, with many applications, from text, audio and video chat or conferencing to streaming audio and video content, requiring low-latency, real-time transmission.

---

15 For example, using the .DAT text data format for SoX, [http://sox.sourceforge.net/](http://sox.sourceforge.net/)
16 Using the standard wave library: [https://docs.python.org/3/library/wave.html](https://docs.python.org/3/library/wave.html)
Pioneering work by Claude Shannon in 1948 (Shannon and Weaver, 1949) established the theoretical foundation for information theory, including the idea of information entropy, which measures the degree of uncertainty in any message. This uncertainty is determined by the nature of the message: if an English message includes the letter q, the next letter is very likely to be u; if a message in English contains ee, the next character cannot be another e; t is more likely to be followed by h than any other letter, and so on. For example, give a sentence in English presented without vowels, it’s possible to recover much, if not all, of the original using our knowledge of English vocabulary, allowed syllables, and in this case English literature to help:

Note that whilst we have saved space here, this is at the expense of some fairly intensive processing to uncompress this text. Shannon’s insight was to recognise that the limits of a communication system were determined not by the message communicated but by the possible messages that could be communicated and their relative likelihood. As Morse had recognised earlier, some letters, such as E and T, are far more frequent than others hence had shorter signals; in contrast not all US-ASCII and Unicode characters are equally likely to occur in a message, and yet each takes the same number of bits to transmit. Shannon’s information entropy uses probability to determine the minimum number of bits needed to communicate a message in a particular system. His original estimate, based on the patterns from looking at groups of eight characters, was about 2.3 bits per character for English, but subsequent work, using longer range characteristics of the language, suggested an entropy of as little as one bit per character (Shannon, 1951).

Users of predictive text systems will be familiar with how quickly smartphones are able to guess the word that is to be typed: these systems draw directly on Shannon’s ideas.

Compression techniques draw on Shannon’s work too, finding clever ways to represent the same (or in some cases, similar) information in a smaller number of bytes.

Huffman coding is closely related to Shannon’s information entropy idea. Rather than using the same number of bits for each symbol, Huffman formalised the idea of assigning shorter codes to more frequently-occurring symbols (Huffman, 1952) in such a way that there would be no ambiguity in decoding.

One simple approach to compression is run-length encoding. If we wish to communicate a message about coin tosses:

```
HHHTTHHTTTTHHTTTTTHHTHTTTT
```

run length encoding shortens the message by simply encoding how many of each symbol there is in each run of it:

```
H3T2H2T3H2T6H2T1H2T4
```

Similarly, our black and white image of Ada Lovelace (Figure 3.17) has very long runs of black pixels with short runs of white pixels: run length encoding would compress this very efficiently.

Another technique involves looking for patterns in the information. In the case of English text, we might simply choose to replace common words ‘the’, ‘is’, ‘to’, ‘of’, ‘and’, and so on, with

---

18 It is a truth universally acknowledged, that a single man in possession of a good fortune, must be in want of a wife.
short, one byte, codes to represent them. A more sophisticated system would be to recursively look for longer and longer patterns in the text, image or audio, replacing these with codes, and storing the pattern against the code. LZW (Lempel-Ziv-Welch) compression (Welch, 1984) takes this approach:

1. Initialize the dictionary to contain all strings of length one.
2. Find the longest string in the dictionary that matches the current input.
3. Emit the dictionary index for the string to output and remove the string from the input.
4. Add the string followed by the next symbol in the input to the dictionary.\(^\text{19}\)
5. Go to Step 2.

For English text, this typically achieves a 50 per cent compression saving. The algorithm most commonly used with the .ZIP compression format is a combination of Huffman coding and an earlier version of LZW compression, which might save 65 per cent of the space for a text file.

In the case of text, scientific data and particularly program source or binary files, any compression has to be \textbf{lossless}: it is essential that we can recover an exact copy of the original information. Huffman codes, run-length encoding, LZW and .ZIP compression all achieve this, and can be applied to any form of data. Other media formats have particular compression algorithms that can be even more efficient, taking account of the particular properties of the medium. For images, .TIFF and .PNG formats both support lossless compression. For audio .FLAC supports lossless compression.

However, in many cases, it is not absolutely essential to be able to uncompress a file to recover all the information originally present: close enough is often good enough. In these circumstances, we can use \textbf{lossy} compression. Very high compression ratios can be obtained but only at the expense of discarding some of the information contained in the original data.

For images, the .JPEG format (Austin, 2009) offers high compression ratios, saving up to 90 per cent of the space required for a full bitmap representation, with little loss of image quality. JPEG makes use of Huffman coding, but it also takes account of how we perceive images – that changes in brightness are noticed more than subtle changes in colour, and that low frequency changes are more noticeable than high frequency ones.

Similar ideas are used for audio compression using the .MP3 format (Sellars, 2000): we notice relatively loud noises more than relatively quiet ones, so the data from the relatively quiet noise can have fewer bits devoted to storing it without significantly impacting our perception of the sound. Similarly transient, high- or mid-frequency sounds capture our attention more than repetitive low frequency ones, and thus deserve more bits for their storage.

We mentioned the idea of creating images and audio using vector graphics or midi notation: these formats are far more compact than high resolution bitmap or PCM audio respectively, as in both cases we store instructions for making the image rather than the image itself. This is related to the idea of Kolmogorov complexity (Kolmogorov, 1998), in which the information of a file or message is measured not by the bits needed to store it but by the bits needed for a program to reproduce it. Take for example the sequence:

\[
\begin{align*}
314159265358979323846264 & 33832795028841971693993751058209 \\
749445923078164062862089986280348253421170679
\end{align*}
\]

Which appears to take 100 bytes to represent as text. An alternative form, using just 35 characters, would be ‘The first 100 decimal digits of Pi’. A genuinely random sequence (such as is needed for a one time pad in cryptography; see page 148) requires as many bytes to describe it as it contains, but a pseudorandom sequence such as might be used in a computer random number generator (that is, one which has similar properties to a genuinely random sequence but is generated by an entirely deterministic system), can be described fully by the program for that generator and its initial seed state.

\[^{19}\] See https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch##Example \textit{for a worked example, qv www.cs4fn.org/internet/crushed.php}
Video is a particular challenge for storing and transmitting: if we simply store a single bitmap at full HD resolution (1920 x 1080 pixels), it needs over 6 mb assuming three bytes per pixel. Video might typically show 25 frames per second, so one minute of video without any compression would take over 9 gb if no compression was used, plus a further 10 mb for uncompressed PCM audio. Obviously such figures are impractical for storing, processing or transmitting video and necessitate some clever uses of lossy compression, including those discussed earlier for images and audio.

Video compression can also make use of the generally static nature of most of what’s seen on screen. Between one frame and the next in a video, relatively little changes, and furthermore our perception generally tunes out things that don’t change much. Thus video standards such as H.264 (Wiegand et al., 2003) need only pay attention to the changing bit of a video signal, perhaps using keyframes a couple of times a second (or for video conferencing, much less frequently) in which all pixels are captured, but then other frames need only store the changes from the previous frame or from the keyframe. For H.264 compression, these techniques reduce the file size for our 1 minute of 1080p video from over 9 gb to around 400 mb; with the later HEVC (H.265) standard (Sullivan et al., 2012), this figure drops to around 200mb. Streaming at a lower resolution would obviously reduce the file size further still.

To give pupils some feel for video compression, they could compare the file size of all the frames in a stop-motion animation to that of the H.264 compressed video exported from a video editor. Telling the same story using scripted animation, such as in Scratch or Blender, would be a useful exercise. Whilst screen cast output from Scratch or rendered output from Blender would produce files of a comparable size, the Scratch program files or Blender project files would be significantly smaller, reflecting the Kolmogorov complexity.

Classroom activity ideas

- Spend time helping pupils develop fluency in converting between binary and decimal numbers, and in doing arithmetic in binary. Counting games, arithmetic drill and practice, worksheets and having pupils record screen cast tutorials are all likely to be useful here. Using functions to convert between binary and decimal, pupils could write their own drill and practice programs in Snap!, Python or other programming languages to practice this.

- Introduce pupils to the idea of encoding and decoding text through Morse code activities, perhaps using torches or electrical circuits to transmit messages, or using the automatic Morse decoders for the BBC micro:bit (20) or Raspberry Pi. (21) Pupils could go on to experiment with converting between text and ASCII or Unicode representations, or develop programs to do so.

- Pupils can explore bitmap images using the tools available in image editing software. (22) They can create small images pixel by pixel in Excel or pixel art editors. (23) Matt Parker’s tool (24) to create Excel spreadsheets from image files is highly recommended. Pupils should construct formulae in the spreadsheet to manipulate the image to desired effects.

- It’s harder to work with audio files, but Audacity provides a simple editor for files in this format, and once pupils’ programming has reached a level of some fluency they can explore creating or editing PCM-encoded audio in Python using the wave library. (25)

Further resources

BBC Bitesize (n.d.) Data representation. Available from www.bbc.co.uk/education/topics/zxnf82


22 For example Photoshop, The Gimp or Pixlr.com,
23 For example http://www.pixilart.net/
24 www.think-maths.co.uk/spreadsheet
25 See also https://people.csail.mit.edu/hubert/pyaudio/ for basic Python audio input / output handling.
Logic Circuits

We discussed the principles of Boolean logic on pages 9. Inside the central processing unit (CPU) that controls the computer, all operations are implemented by using logic gates to switch the bits that make up digital data between different sets of logic circuits. The programme of study for Key Stage 3 expects pupils to have some familiarity with how logic gates can be used in circuits:

understand simple Boolean logic [for example, AND, OR and NOT] and some of its uses in circuits

We can introduce pupils to the AND, OR and NOT gates through simple electrical circuits (Figures 3.18–3.20):

![Figure 3.18 Circuit to illustrate an OR gate – LAMP1 lights if SW1 OR SW2 is closed](image)

![Figure 3.19 Circuit to illustrate an AND gate – LAMP1 lights if SW1 AND SW2 are closed](image)
With somewhat more authenticity, these gates can be built from individual transistors, perhaps on a breadboard (Figures 3.21–3.23):

Before the invention of the integrated circuit, digital computers would be made using logic gates composed of surface-mounted, transistor-based circuits such as these.
The beauty of abstraction though is that we don’t need to worry about the internal operation of logic
gates; we can treat them as ‘black boxes’ which produce certain output for certain input according
to their truth tables.

Thus we can create simple logic circuits using individual logic gates as the components, rather
than thinking about switches or transistors.

For example we can build (or simulate) the circuits (Figures 3.25–3.26):

![Figure 3.25 NOT (A AND B)](image)

and:

![Figure 3.26 (NOT A) OR (NOT B)](image)

Using a symbolic representation of the gate, rather than its internal structure. Note that the two
circuits above have the same truth tables.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

And thus the two are functionally equivalent. A gate with this truth table is called NAND (NOT AND).

More complex circuits can be constructed. Thus for the truth table:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Carry</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

We have Carry = A AND B and Sum = (A OR B) AND NOT (A AND B) (Figure 3.27):

![Figure 3.27](image)
If we replace False with 0 and True with 1, the table becomes:

<table>
<thead>
<tr>
<th>A + B</th>
<th>Carry</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 + 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 + 1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1 + 0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1 + 1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Which allows binary addition, at least for just one bit, to be implemented using nothing more than logic gates. This circuit is known as a half-adder.

Two half adders can be combined using an OR gate, to allow three inputs, one from a previous carry, plus two ‘bits’ of data, producing a sum and carry for bit wise addition (Figure 3.28):

![Figure 3.28 Full adder](image)

A set of eight of these, connecting the Carry output of each to the Previous Carry input of the next would allow two bytes to be added together.

More complex logic circuits still can be designed and built. Given any truth table, it's possible to construct a logic circuit that will produce the desired output using just a combination of NOT, OR and AND gates. All logic circuits, including NOT, OR and AND gates, can be built using NAND gates.

- Circuit simulators such as logic.ly, circuitlab.com and logic lab allow pupils to experiment with electrical, electronic and logic circuits on screen. It's well worth giving pupils some experience of creating simple logic circuits using switches, transistors or integrated circuits if the resources are available.
- You can get pupils themselves to act as a logic circuit, some taking on the role of gates, others the part of bits. See this BBC clip of implementing a two bit adder: [www.bbc.co.uk/programmes/p01m5xfs](www.bbc.co.uk/programmes/p01m5xfs)
- Give pupils particular scenarios to create logic circuits: home security systems and traffic management are popular contexts.
- Pupils can build simple logical circuits out of redstone in Minecraft.

**Classroom activity ideas**

- Provide pupils with increasingly sophisticated logic circuit diagrams, asking them to work out the truth table by tracing the output at each gate for all the possible combinations of TRUE and FALSE inputs to the circuit. A more challenging problem is to create the logic circuit for a given truth table.


28 [www.neuroproductions.be/logic-lab/](www.neuroproductions.be/logic-lab/)
Hardware Components

One quite intuitive and generally helpful way of thinking about computers is as machines which accept input, process this according to some stored set of instructions and produce output (Figure 3.29).

![Figure 3.29](image)

In thinking about hardware components, it’s worth considering each function here separately.

**Input**

The form of input will vary, depending on the type of computer and the uses to which it is put. On a laptop, you might typically find a keyboard, a trackpad, a microphone and a webcam, as well as ports into which other input devices, such as a USB mouse, could be plugged in. On a smartphone, you would probably find a touch-sensitive screen, some other buttons, a microphone or two, a couple of cameras, a global positioning system (GPS) receiver, an accelerometer, perhaps a barometer and again one or more ports for additional input devices.

Both Raspberry Pi and BBC micro:bit have support for a broader range of sensor input, including simple switches and the digitisation of analogue signals connected to their general purpose input / output (GPIO) pins or connector. Sensors can be connected to these pins which can, for example, be used to capture temperature levels over time, or proximity in some robot control applications. With other computers, an interface can be used to provide similar functionality.

For trackpads, touch screens, microphones and cameras it’s necessary for the computer to convert the continuous, analogue real-world data into a digital format before it can be processed, stored or transmitted by the computer. As discussed on pages 92 - 94 in the context of images and sound, digitisation inevitably involves throwing away some of the fine detail of the real-world information.

Some pre-processing can be applied to the raw input signals, for example voice recognition provides an alternative form of input using the microphone, or positions of objects in three dimensions can be determined using two cameras or a linked laser and sensor for establishing depth, as in Microsoft’s Kinect.²⁹ A basic form of brain–computer interface is possible using current technology, typically through sensing electrical activity in the brain.

**Processing**

The fetch–decode–execute cycle in which processors execute machine code instructions is described on pages 44 - 45. Programs written in high-level languages are converted into machine code using interpreters or compilers – most of the programs that run on the computer are already compiled as machine code binaries, and so this process is typically hidden from the user. The machine code instructions and the data on which the programs operate are all stored together in memory (see later) and the computer provides a fast way of moving both program instructions and data out between the processor and memory, the internal (or ‘front side’) bus.³⁰

Processors work very, very quickly. Modern operating systems are efficient at managing the load on the processor so that, despite giving the appearance of always being ready for whatever instruction it is next required to execute, it can also run instructions from many of the other programs on the computer almost simultaneously. This is called multi-tasking.

The processors inside modern computers are typically multi-core chips, containing perhaps four (or more) CPU cores and cache memory (see later), each operating independently, capable of running instructions from quite separate programs.

³⁰ Neil Brown explains that modern processors use optimisations such as caches, pipelining, out-of-order execution, speculative execution and microcode, in addition to the basic fetch–decode–execute cycle: [https://academiccomputing.wordpress.com/2012/04/29/the-computer-is-a-lie/](https://academiccomputing.wordpress.com/2012/04/29/the-computer-is-a-lie/)
literally at the same time. Multicore processing isn't confined to the desktop or laptop form factor either: some current smartphone models have eight-core processors.

As well as the main, typically multicore, processor modern computers often have other processors, most notably graphical processor units (GPUs) now often designed specifically for the parallel processing needed for physics simulation and fast 2D and particularly 3D rendering, and most commonly deployed for video games (Sony’s PlayStation 4 has some 1,152 parallel cores for its GPU). For the iPhone 5s, Apple introduced a ‘motion’ coprocessor (the M7) alongside the smartphone’s main CPU, dedicated to processing sensor data and running on low power even when the phone was asleep.

Current developments include massively parallel processing, in which complex computing problems are shared between many processors with results being subsequently combined, as well as processing ‘in the cloud’, in which complex tasks are handled not by the user’s own computer but by running programs on computers to which it communicates via the internet. Speech to text processing of services such as Apple’s Siri is accomplished in this way.

The fastest memory available will be the registers and ‘level 1 cache’ memory built into each CPU core itself, but even with modern processors the amount of data that can be stored here is very limited. Elsewhere on the CPU chip, and connected to the rest of the computer via the bus is the level 2 cache – larger than the level 1 cache, and somewhat slower, but still much faster to access than the main memory.

On the internal bus, and connected directly to the main circuit board of the computer (the motherboard) is fast, high capacity memory called RAM, although this is typically ‘volatile’, meaning it loses all the data stored in it when the computer is switched off.

Rather slower, but again of much larger capacity will be the computer’s main drive: until recently, this would typically contain magnetic disks able to retain data when the power is turned off, but these are increasingly being replaced by faster, non-volatile ‘flash’ memory or solid state drives (SSDs), similar to what you might also find in USB sticks or the memory cards used for digital photography.

Optical storage, such as CD-ROMS, DVDs and Blu-ray disks, is slower still, but costs for these media are low, making them suitable for long-term storage of data rarely needed for processing. These media are not, by modern standards, of particularly high capacity and thus many find the need to connect high-capacity external hard drives or SSD drives to the computer, perhaps via USB or a higher bandwidth external bus interface.

Storage capacity has become larger and larger even faster than processing has become faster (Walter, 2005), and thus now data centres connected via the internet can provide very high capacity, but relatively slow storage for data at a very low cost. Whilst once, long-term, archival storage might have made use of magnetic tape, there are interesting developments using low cost, high capacity hard drives for this sort of ‘cold’ storage.

31 See, for example, www-01.ibm.com/software/data/infosphere/hadoop/mapreduce/
32 For example Google Compute, Microsoft Azure and Amazon EC2.
33 www.extremetech.com/extreme/210492-extremetech-explains-how-do-ssds-work
Internet pioneer Vint Cerf has, perhaps surprisingly, argued that the best approach to very long term (for example, millennium-long) archival storage might more reliably use paper than digital media, given that no special systems need to be maintained to ensure that paper remains readable.

**Power and cooling**

Other internal hardware components are needed too.

Processing requires power, and so one of the most noticeable components inside a desktop computer will be the power supply unit, providing regulated current at the various voltage levels needed for the computer’s components. In the case of business critical machines some form of alternative power supply, such as an uninterruptible power supply (UPS; essentially a large battery and associated monitoring sensors and software) or perhaps even an emergency generator may be necessary. Battery life remains a problem for portable technology including laptops, tablets and smartphones, although battery technologies have advanced significantly in recent years alongside advances in processor and storage efficiency and better power management software at operating system level. A relatively recent development has been the use of additional, external batteries to provide a top-up charge for the smartphones on which many have come to rely. Convenient access to a cheap source of power is one of the principal considerations when siting a data centre.

The processing that computers do generates lots of heat (and thus the 2nd law of thermodynamics holds, as entropy increases overall). Processors stop working if they get too hot and thus attention must be given to keeping processors cool. In a traditional desktop computer, CPUs are mounted with passive heatsinks and active fan cooling, and a smaller version of these components will also feature on laptop computers. Tablet computers and smartphones typically do not include active cooling with a fan although much care is taken in their design to ensure effective passive cooling. Bare board computers such as the Raspberry Pi and BBC micro:bit achieve sufficient cooling in normal use as air can freely circulate over the processor. Cooling becomes a particularly important consideration in building data centres, where a large number of processors and other components are packed together in a small space.

**Output**

Computers are able to produce many different forms of output. On a laptop or desktop computer these are likely to be the screen and speakers, together with connections for external peripherals such as printers or headphones.

On a smartphone, tablet or games console controller, outputs might also include a small motor to produce vibrations. Smartphones typically include bright light emitting diodes (LEDs) used as a flash for photography or to provide extra light when recording video.

Other output devices can be connected too, for example the computer can be used to control motors, such as in a robot. In addition to traditional 2D display technologies, computers can also power virtual or augmented reality headsets: the former replaces the wearer’s view of the world with a 3D computer generated image (shown as different images to each of the wearer’s eyes), the latter adds an additional layer of information to the wearer’s view of the world.

Similar to the way in which a computer can print information on paper through sending instructions to a printer, computers can send instructions to 3D printers, producing 3D objects through detailed layering of a plastic resin (or other material) according to the instructions received.

**Connectivity**

The traditional model of input – processing – output has evolved to include network connections. The data a computer processes need not be provided directly through input devices attached to the machine, it could easily come via one or more network connections. Similarly the information the computer outputs could be transmitted via a network connection.

---

For example, a typical web-server is unlikely to have a keyboard or screen connected to it directly – it accepts requests for web pages or commands via its internet connection and responds with the HTML for the page or other output via its internet connection.

A smartphone includes a number of different network connections, from near field communication (NFC) for passing small packets of data such as cashless payments over very short distances, through Bluetooth for external keyboards and hands-free audio, WiFi for high-speed internet access, through to the longer range connection to the phone network for voice and data, possibly at high speed using 3G or 4G systems. WiFi and Bluetooth connectivity is typically provided as standard on laptop and tablet computers, and is also built in to the Raspberry Pi 3. The BBC micro:bit includes Bluetooth connectivity. Traditional cabled network connections provide greater communication bandwidth, and are less likely to suffer from contention issues than the WiFi equivalents.

Many devices now contain microprocessors and should be thought of as small, dedicated use computers with their own input / output systems: for example a digital thermostat includes a heat sensor, some form of control interface for user input, processing capabilities and a stored program, a display and a control interface for the heating system itself. Increasing numbers of these devices now provide network connectivity too, most often via WiFi, but perhaps via Bluetooth or the mobile phone system, becoming part of the ‘internet of things’. Such connectivity provides greater convenience, such as the ability to turn on central heating via smartphone on the journey home or to upload photos directly to the internet from a digital camera, but also allows integration with other internet-based systems and cloud-based processing. For example, a smart thermostat could ‘learn’ typical use patterns and adjust these predictively to take account of weather forecasts, or a smart, internet-connected refrigerator could autonomously submit orders to an online supermarket as staple supplies run low. Many are concerned over the privacy and security issues associated with such systems.

Classroom activity ideas

- Show pupils how input and output devices and the network are connected to a computer. Disassemble a computer as pupils watch, explaining the purpose of each of the components in turn. Explain that the components pupils see are made up of smaller components, to illustrate the multi-layered nature of abstraction in computing. Some teachers make displays of computer components.
- Compare the components of different types of computer system, showing how the internal components of desktop, laptop, smartphone and Raspberry Pi computers must all accomplish the same function, but that these different form factors mean that their physical forms can be quite different.
- Demonstrate CPU usage using operating system tools to monitor activity (Performance Monitor for Windows, Activity Monitor on OS X and the top command on Linux).
- Pupils in an extracurricular computer club could perhaps build their own computer from component parts, or fix a broken computer.

Further resources


Google Data Centres (n.d.) Available from www.google.co.uk/about/datacenters/

Ifixit (n.d.) Repair and ‘tear down’ guides to common devices, including some great photos of internal components. Available from https://www.ifixit.com
Software Components

None of this computer hardware would do anything if it had no software: the programs that make it work, that make it useful. The many layers of software that make up a computer system are each abstractions of the software systems, and ultimately the hardware, beneath them. At each layer of the system, assumptions are made about how the layer lower down behaves, without needing to know how it works, which means that:

- There’s no need to deal with the complexity of the system at that layer since this has already been addressed in a reliable way.
- There’s no duplication of the functionality provided by the lower layer, as this is done once for all the different programs running at the upper layers.
- The internal operations of the lower layer are generally something with which we don’t need to concern ourselves; indeed in many systems the internal operation of lower layers is deliberately hidden from those working at upper layers, sometimes for reasons of security, but also for proprietary, commercial reasons.

Working from the bottom of the software stack up, and thinking in terms of a general-purpose computer such as a laptop, desktop, smartphone, tablet or Raspberry Pi, we typically begin with a very small set of firmware instructions, compiled into machine code, which provide just enough functionality to load and run (to ‘bootstrap’) an operating system. Many of the internal components of the computer, such as disk drives, will have their own firmware too, and modern CPUs often include a layer of ‘microcode’, which is essentially an interpreter for machine code which sits between the hardware itself and the abstraction of the internal architecture that’s presented to the rest of the system.

Operating systems (OS) such as Windows, OS X, Linux, iOS and Android sit between the user’s experience of the computer and the computer hardware itself. Operating systems are themselves multilayered. One level deals with fundamental operations, from managing multi-tasking for the different system and user programs to be run on the CPU, main memory and longer term storage, input, output, network connectivity, power and cooling, and presenting abstractions (such as a file system, virtual memory and inter-process communication) of all these complex subsystems in a simple, consistent and reliable way to other programs on the system.

The various hardware components typically have device driver software, running at operating system level, although not formally part of the operating system, which allow components from different manufacturers and with different specifications to be used by the operating system without the OS having to deal with the specifics of implementation.

Outside of the operating system itself there are often a number of utility programs that are useful, if perhaps not absolutely essential to the operation of the computer, such as programs to find files, to protect against viruses and other malware and to manage the installation of other programs.

The operating system also provides one or more user interfaces – a way for the user to interact with the computer’s core functions and with other software that might be running on it. The most basic of these is a command line interface (CLI), in which commands can be typed on screen and responses displayed on screen. Some computers (for example Linux servers) start up in this mode, but more often you can access this interface through launching a shell or terminal program. Whilst seeming to lack the functionality, familiarity and ease of use of more familiar graphical user interfaces (GUIs; see below), this stripped down way of interacting with computers offers power and flexibility, and can sometimes...
be the only, or more often the fastest, way to get things done.

The other layer of a modern operating system is the **graphical user interface (GUI)**, which provides a consistent look and feel for the user’s interaction with the computer in a way that’s more intuitive and easier to learn through exploration than the CLI. The GUI provides a way to interact with the computer through keyboard, mouse and now voice input, task switching, an interface to manage files and folders, control interfaces for sound, networking, dates and times, and so on, as well as managing windows for programs, displaying text and graphics, looking after the printer and providing accessibility support such as voice synthesis. It’s typically the GUI that people think of when talking about Windows or Mac (OS X) operating systems. All of this of course needs processing capacity in order to operate, and so computers running as servers or in data centres would typically not run, or even have, a GUI.

Whilst the CLI or GUI allow us to interact with the computer and run utility programs if needed, they don’t, in themselves, provide software for getting useful work done. For this, we use application software and most of the programs which we think of as running on a computer fall into this broad category. Within this category are, for example:

- **Office productivity programs**: such as a word-processor, spreadsheet and presentation software, desktop publishing software, calendars, task management, contact management.
- **Media production tools**: such as image editors, drawing programs, audio editors, music composition software, video editing software, 3D animation packages.
- **Media management tools**: for example music players, video players, image galleries or photo browser.
- **Communication software**: a web browser, email client, video conferencing tools, instant messaging.
- **Technical software**: computer algebra systems, bibliography management tools.
- **Games and educational software**.

Software in all of these categories can run on traditional Windows, OS X or Linux desktop computers or laptops, but these categories of software are also available as ‘apps’ (short for application program) on iOS or Android smartphones and tablets, and can be run on remote web-servers and accessed using just a web browser via the internet (as is the case with Google’s Chrome operating system).

A few special categories of application software deserve particular mention:

- **Server software**: via the internet (or perhaps just a local network), one computer can run a program to provide services to many other computers, for example managing user accounts and password authentication on a school network; saving files centrally; storing, forwarding and providing access to email; serving static or dynamically-generated web pages when requested (see page xx), managing a database of records so that connected computers can each update or interrogate a single consistent version of the information contained in it, and so on.

- **Programming language software** and associated tools: converting a program written in a human-readable, high-level language such as Scratch or Python requires an interpreter or compiler; although once compiled the program could be treated as any other application or system program. Programmers also use a range of tools to support the process of writing programs, including text editors or integrated development environments (such as the Scratch web and offline editors or Python’s IDLE).

- **Virtualisation**: theoretically, one computer system can simulate any other: in practical terms this allows computers to run virtual emulations of other computer systems. Programs such as Virtual Box or VMWare provide a simulation of computer hardware, running as software, onto which other operating systems can be installed: this is a great way to explore other operating systems safely, cheaply and securely, and also offers a useful way to deploy preconfigured combinations of operating systems and application software for particular purposes.

As with other application software, these categories can be used on Windows, Mac (OS X) and Linux operating systems, or accessed remotely via the web. With the exception of providing some simple web-based services to the local network, support for these categories is at present largely absent from iOS or Android platforms.
Classroom activity ideas

- Use the operating system tools to monitor activity (Performance Monitor for Windows, Activity Monitor on OS X and the top command on Linux) to show pupils all the programs which run on the CPU to demonstrate how the operating system manages multi-tasking.
- Provide pupils with the opportunity to try out several operating systems, perhaps setting them the same task (sending an email, making a presentation, writing a short program) to accomplish using the Windows GUI, Linux at the command line, a smartphone and just a web browser.
- Give pupils the opportunity to install an operating system from scratch, perhaps using virtual hardware. If using open source software such as Linux, pupils could then assemble and test a suite of application programs selected with a particular user in mind.
- Work with your network manager to ensure pupils get some experience of working with a command line interface, perhaps using the command prompt on Windows or a Linux shell via terminal access or virtualisation.

Further resources

BBC Bitesize (n.d.) Software. Available from www.bbc.co.uk/education/guides/zczxqr82/revision and www.bbc.co.uk/education/guides/z6r86sg/revision, also operating systems: www.bbc.co.uk/education/guides/ztdctfr/revision


Stephenson, N. (1999) In the beginning... was the command line. New York, NY: Avon Books.

Physical Computing

In order for a computer to be able to do anything with the real world, it needs some form of input to get data in, and some form of output to put information back out.

Traditional 'control' activities certainly have their place in the new computing and Design & Technology programmes of study.

Teaching control technology is perhaps implied by the requirement that pupils be taught to:

- understand the hardware and software components that make up computer systems, and how they communicate with one another and with other systems

More importantly, it is specifically required in the Key Stage 3 design and technology programme of study:

- apply computing and use electronics to embed intelligence in products that respond to inputs [for example, circuits with heat, light, sound and movement as inputs and outputs]

The first of these statements provides some opportunity for pupils to engage practically with the ideas of Boolean logic circuits. The second offers ample scope for pupils to engage fully with physical computing and making.

Perhaps the easiest way into the realm of physical computing is through using computers to monitor activity in the real world. A very simple introduction might involve recording or plotting the level of noise in the classroom using Scratch’s microphone input (Figure 3.30):
The Makey Makey interface board\(^{35}\) plugs into a computer’s USB port allowing other conductive objects to function as a replacement for some of the keys on the computer’s keyboard. This can be combined with pupils’ own Scratch programs to produce, for example, a maze game controlled by jumping in buckets of water or a piano with keys made from bananas. The PicoBoard\(^{36}\) provides an alternative input interface for Scratch.

Integrating monitoring software with web-based services allows pupils to explore some aspects of the ‘internet of things’, for example setting up a bird box camera which uploads a photograph to the web when a bird enters or leaves the box, or tweets data from a school weather station in response to particular queries.

There are some great cross-curricular opportunities here, for example, the use of dataloggers in science experiments, activity monitors in PE or weather station data\(^ {37}\) for science and geography. Working with real-world data such as this provides a very motivating context for visualisation and exploratory analysis, and perhaps some scope for introducing ideas of machine learning.

Beyond monitoring activities, many teachers report success with having pupils write software which controls real world components. The ‘Hello, world’ of this sort of programming is typically flashing an LED on and off (see Figure 3.31).

Integrating monitoring software with web-based services allows pupils to explore some aspects of the ‘internet of things’, for example setting up a bird box camera which uploads a photograph to the web when a bird enters or leaves the box, or tweets data from a school weather station in response to particular queries.

There are some great cross-curricular opportunities here, for example, the use of dataloggers in science experiments, activity monitors in PE or weather station data\(^ {37}\) for science and geography. Working with real-world data such as this provides a very motivating context for visualisation and exploratory analysis, and perhaps some scope for introducing ideas of machine learning.

Beyond monitoring activities, many teachers report success with having pupils write software which controls real world components. The ‘Hello, world’ of this sort of programming is typically flashing an LED on and off (see Figure 3.31).

---

35 www.makeymakey.com/
36 www.picocricket.com/picoboard.html
37 See, for example, www.raspberrypi.org/blog/school-weather-station-project/
38 https://vimeo.com/4313755
on Raspberry Pi’s sense HAT,[39] or create a digital musical instrument.[40]

Pupils might take these ideas of monitoring and control and apply them to projects involving robotics. For example, pupils can build a model or robot out of Lego NXT or EV3 Mindstorms kit, incorporating sensors and motors, and then write code in Enchanting Scratch or EV3’s own programming language to control how their model moves in response to the input signals received from the sensors.

One way of thinking about a robot is as a computer which can move, perhaps as a single, integrated system following a sequence of instructions such as the Pro-Bot, Bigtrak or a floor turtle, as a flying drone under the direct control of its operator, or as a largely stationary device with one or more motors controlling moving parts, such as a robotic arm under computer control used in industrial manufacturing or a surgical robot under the remote control of a human operator.

Robotics has long had wide applications in industry, where repetitive tasks can be performed effectively and efficiently by machines, but as better, or ‘smarter’ algorithms have been developed by computer scientists, more and more decision making capabilities can be built in to the robot, so that the robot is able to autonomously react to changes in its environment. Autonomous, self-driving cars such as those pioneered by Google are an example, even if we may think of them as cars rather than robots. The long communication delays between Earth and Mars mean that the robotic Mars rover[41] must use lots of event-driven, when...do programming to be able to respond to what happens in its environment without waiting for control instructions from Earth. One application of machine learning will be programming robots to respond to input data to improve their own operation over time, perhaps particularly in such far-flung settings.

Whilst few might attempt such projects with a whole class, an advanced group of pupils or an extracurricular club might combine Design & Technology and computing skills, knowledge and understanding to build and program their own robot, perhaps entering a competition with their work or focussing on developing a solution to a real-world problem.

There are many platforms available on which pupils can develop their understanding of physical computing:

- Interface boards such as Makey Makey provide an easy way to connect the ‘real world’ beyond keyboard and mouse to a computer and can be used directly with lots of different programs.
- Small microcontroller based boards such as CodeBug and Crumble allow pupils to write programs on screen and then flash (download) them to the board to run independently; both of these examples can be used for both monitoring and control.
- Lego WeDo hardware provides simple sensors and a motor, and is connected to a computer or tablet via Bluetooth; it can be programmed using Scratch as well as its own tile-based language.
- Lego Mindstorms is a more sophisticated system, with a wider range of sensors and motors. Programs are downloaded to the Mindstorm control brick and can then run independently of the computer on which they were written.
- Arduino microcontrollers provide a range of boards able to work with a variety of different components. Again programs are written on one computer and then downloaded to the Arduino board.

The two most common platforms for physical computing activities at present are the BBC micro:bit and the Raspberry Pi, both of which have been mentioned elsewhere in this guide, but both merit further discussion here.

### Raspberry Pi

The Raspberry Pi is a small, cheap and high-powered bare circuit board general-purpose computer. It was created by a small, Cambridge-based team who had noticed the decline in undergraduate admissions to university computer science courses and believed that easy access to a computing platform that positively encouraged tinkering, programming and making would be helpful in supporting computing education. The first model went on sale to the general public in 2012.

---

41 [http://mars.nasa.gov/mer/overview/](http://mars.nasa.gov/mer/overview/)
The Raspberry Pi foundation sees its mission as:

> to put the power of digital making into the hands of people all over the world, so they are capable of understanding and shaping our increasingly digital world, able to solve the problems that matter to them, and equipped for the jobs of the future.

Figure 3.32 Raspberry Pi 3 (CC by-sa Herofungus)

Version 3 of the Raspberry Pi (Figure 3.32) is a capable machine: the board provides four USB inputs, such as for mouse and keyboard, HDMI video and audio output to connect to a monitor or television screen, Bluetooth, WiFi and wired networking, a connector for a camera module (sold separately), and a set of general purpose input/output pins used for the control and monitoring needed for physical computing. The processor is a four-core ARM chip running at 1.2 GHz, similar to that used in smartphones, and it is powered by a micro-USB connector. It has 1 gb of RAM but has no built-in permanent storage, instead the operating system, application software and any data are stored on a removable micro SD card, from which it boots.

The Raspberry Pi can use a number of different operating systems, including a version of Windows 10, and Acorn’s RISC OS first released in 1987 and popular at the time in UK schools. Most users however use a version of Debian Linux, Raspbian that has been tailored to both the Raspberry Pi hardware platform and the foundation’s educational mission. Raspbian has an easy-to-navigate desktop GUI using a similar system of icons and menus to other operating system GUIs.

The bundled software focusses, unsurprisingly, on programming but includes a broader range of application software too: there’s the Libre Office suite, a web browser and an email client, as well as the high-end computer algebra system Mathematica.

On the programming side, we have Python with the IDLE IDE (integrated development environment), a bespoke version of Scratch with support for the GPIO pins and Sonic Pi, a Ruby-like language for composing (and performing) music. Raspbian includes Minecraft as standard, with the API (application programming interface) to allow Minecraft to be controlled through programming in Python. Many additional programs can be installed very easily: for example, the command (typed in the shell):

```bash
sudo apt-get install tree
```

is all it takes to install the tree utility.

The Raspberry Pi Foundation has assembled a collection of curriculum resources, many of which are directly relevant to the Key Stage 3 computing curriculum, with others providing much that might inspire pupils in extracurricular computing clubs within or beyond school.\[^42\] The Foundation merged with Code Club towards the end of 2015. Whilst the Code Club activities\[^43\] have been written with primary pupils in mind, many, particularly those on HTML and Python, would be appropriate for Key Stage 3.

The Raspberry Pi Foundation has developed two-day face-to-face PiCademy workshops. Day 1 is spent learning about computing and the Raspberry Pi, including physical computing, Minecraft and Sonic Pi. On day 2, participants work in teams, with contributions from the Raspberry Pi team to develop their own project ideas. The Raspberry Pi community also hosts frequent local ‘Raspberry Jam’ events, in which community members meet to share their knowledge, learn new things and show off what they’ve done with their Raspberry Pi.

**BBC micro:bit**

Back in the 1980s the BBC played a pivotal role in the early days of computing education. The BBC sponsored the development of a home computer by Acorn, the BBC Micro and developed television content to promote computer literacy in the home.

The BBC Micro was chosen as one of the computer models that would be supplied to every school.

Building on this legacy, and as part of a year long ‘Make it digital’ initiative across the BBC’s media, the BBC drew together a consortium of some 29 partner organisations (including ARM, Samsung, Microsoft and Lancaster University) to develop the BBC micro:bit, with the aim of inspiring ‘young people to get creative with digital and develop core skills in science, technology and engineering’.

In 2016, close on 1 million micro:bits have been distributed via schools to the entire Year 7 national cohort. Significantly, the micro:bits, whilst distributed via schools, are intended to be given to this cohort of pupils themselves. School, pupils and others are able to buy further micro:bits if they desire.

The micro:bit (Figures 3.33–3.34) is smaller than the Raspberry Pi and, unlike the Raspberry Pi, it cannot be programmed directly, but rather must have programs downloaded to it from a connected computer, tablet or smartphone.

The hardware includes an ARM microcontroller, running at 16 MHz, 16 KB of RAM and 256 KB of flash memory in which programs are stored. Input is through two buttons, an accelerometer, a magnetometer and the GPIO edge connector. Output is through a 25 pixel display or via the GPIO connector (for example a speaker can be wired between connectors 0 and GND to produce simple audio). Connectivity is via USB and Bluetooth, as well as, somewhat unexpectedly, the GPIO pins.

Programming the micro:bit is done via a web-based interface at www.microbit.co.uk/create-code which provides access to four different editors: a block-based javascript-like editor from Code Kingdoms, a blockly-based block editor, Microsoft’s TouchDevelop and an online version of the Mu educational IDE for microPython (see Figures 3.35–3.36).

Figure 3.33 BBC micro:bit showing buttons, 25 pixel display and IO connector: CC by-sa Gareth Halfacree

Figure 3.34 The other side of the BBC micro:bit showing power and data connectors, processor, accelerometer, compass and reset button

Figure 3.35 Simple micro:bit dice program in Blocks

Figure 3.36 The same program converted to TouchDevelop

45 Alternative programming platforms are available such as Microsoft’s www.pxt.io/
When the micro:bit runs.

```javascript
function onStart( ) {

}

function onShake( ) {
    microbit.say(Random.number(1, 6));
    wait(1000);
    microbit.clear();
}
```

A similar program in CodeKingdoms' Javascript editor (code view)

```javascript
from microbit import *
import random

while True:
    if accelerometer.is_gesture("shake"):
        display.show(str(random.randint(1, 6)))
    sleep(1000)
```

A similar program in microPython

There are a couple of particularly nice features of the micro:bit code editors. Firstly, there’s an on-screen emulator, so (with the exception of microPython) it is possible to test your code on screen without downloading it to the micro:bit (Figure 3.37).

Figure 3.37 On-screen micro:bit emulator running the above program

Secondly, although these editors work in the web browser, they don’t need any server side processing to work: compiling the program you write happens inside the browser itself, so once they have been accessed online they can be subsequently used offline. Micro:bit source code files can be uploaded or downloaded to the editor from the desktop.

Once you have written your program, to run it on the micro:bit itself you first compile the program, which produces a .hex machine code file. This contains a pre-compiled run time environment, a device abstraction layer that sits between the hardware and your program, and your program compiled into ARM mbed machine code. Connecting the micro:bit to your computer via a USB cable, it shows up as if it were a USB memory stick, so you can simply drag the compiled .hex file across onto the micro:bit. A quick press of the reset button and your code should run very happily on the micro:bit itself. You can now plug in the micro:bit’s own battery pack and disconnect the USB cable.

As you would expect, there’s a good range of support materials available from the BBC themselves and many of the other partner organisation. The project site at [www.microbit.org/](http://www.microbit.org/) is the best place to get started.

CAS (Computing At School) regional centres and master teachers have been active in supporting teachers with introductory CPD (continuing professional development) to the micro:bit, as have many of the partner organisations.

### Classroom activity ideas

- Pupils could use the micro:bit, Raspberry Pi Sense HAT or GPIO pins, or a Makey Makey to control a simple game.
- Pupils might use sensors to collect weather data for school and then analyse this for patterns, relationships and interesting exceptions.
- Ask pupils to write a program which could control a set of traffic lights in the correct sequence, perhaps on screen initially. Can they connect suitable LEDs to the computer and control these directly?
- Pupils could build and program a robot that could find its way out of a maze.

### Further resources


Available from [www.bbc.co.uk/education.guides/zdsbwmn/revision](http://www.bbc.co.uk/education.guides/zdsbwmn/revision)


Technology will Save Us (n.d.) Available from www.techwillsaveus.com/

References


