binary ones (on) or zeros (off), and the entire contents of the screen can be regarded as a 'mapping' into dots of the bits that comprise those bytes of screen RAM. Unfortunately, although the BBC Micro, the Spectrum and the Commodore 64 all use this mapping technique, none of them does so in a straightforward manner. For our purposes, the simplest method would be to divide each row of the screen into pixel bytes numbered consecutively from left to right, the leftmost byte in a row following the rightmost in the preceding row. For a variety of reasons this is not the case on any of these machines. Let's consider each case separately.

The Spectrum screen is always in high resolution mode, and a fixed area of memory is set aside for mapping the screen. The mapping is complex, however, as the screen is divided horizontally into three blocks of eight PRINT rows, and each print row is divided horizontally into eight pixel rows. The addressing of the bytes that comprise these rows is sequential within the rows, but not between the rows. The BBC Micro and the Commodore 64 do not follow this pattern, but are equally devious. For the moment, it is considerably easier to understand if we confine ourselves to outputting ASCII characters to the screen.

This is something that the machine does all the time, and there are, therefore, machine code routines in ROM for the purpose. Given a suitably detailed description of their operation, we can call these routines from our own Assembly language programs. What we need to know is the call address, the communication registers, and any necessary preliminaries.

On the Spectrum there are no preliminaries to observe, and the communicating register is the accumulator, which must contain the ASCII code of the character to be printed. We need only issue the instruction RST $10 and the character whose code is in the accumulator will be printed on the screen at the current cursor position. This is very much the pattern of the other two systems, but the RST (ReSTart) op-code is peculiar to the Z80 command set: it is a single-byte zero-page branch instruction that must take one of only eight possible operands—$00,$08,$10,$18, etc. to $38. Each of these locations points to the start address of a ROM routine, somewhere in zero page. These routines are typically dedicated to handling input and output, and we call them through the RST instruction rather than directly by address. This is partly for speed (it is quicker to use RST than CALL, although only the CPU would notice the difference), and partly for the sake of the program's portability. If every Z80 programmer knows that RST $10 calls the PRINT routine on every Z80 machine, then nobody is going to bother about where a particular systems software engineer actually locates the PRINT routine, and the engineer is free to locate it anywhere, provided that zero page is arranged in such a way that the RST locations direct programs to the start addresses of the commonly-agreed routines.

On the BBC Micro the procedure is similar: an ASCII code in the accumulator combined with a JSR $FFEE command will cause the character to be PRINTed on the screen at the current cursor position. This is the OSWRCH routine, much referred to in BBC literature and well documented in the Advanced User Guide.

The Commodore 64 follows the pattern of the other two machines. An ASCII code in the accumulator and a JSR $FFD2 command causes the character to be PRINTed at the current cursor position. This is the CHKOUT routine, and is documented in the Programmer's Reference Guide.

This, therefore, is the general pattern of use of ROM routines and demonstrates the principle of communication registers. A communication between the calling program and a subroutine may pass either way — an input routine, for example, might pass a character from an external device to the CPU via the accumulator. Even when there is no substantive information passed like this, an error code may well be returned from the subroutine through one of the registers. This sort of protocol is documented in the many machine-specific works of reference now available.

Input from the keyboard and other devices will be dealt with in later instalments, as will high resolution plotting from machine code. We conclude this instalment of the course with a summary of the various aspects of Assembly language and machine code programming.

IN SUMMARY

We began the course with a wide-ranging look at machine code from a very non-specific point of view, trying to dispel some of its mystique and place it in context as just one kind of code among all the others that we (and computers) use. We have seen how the same sequence of bytes in RAM can be interpreted at one moment as a string of ASCII data, at the next as a BASIC program line, at the next as a string of two-byte addresses, and then again as a sequence of machine code instructions. A few minutes spent playing with a machine code monitor program should convince you that some sequences of bytes can be disassembled as three quite different, but valid, sequences of instructions — depending on whether you start the disassembly at the first, second or third byte in the sequence. Nothing intrinsic to the code prevents this happening, and the CPU itself cannot tell whether it's executing the code that you wrote, or some garbled version of it, accidentally transposed in memory.

We went on to consider the organisation of memory, and the common conventions of addressing. To make any sense of this we had to begin the study of binary arithmetic, which immediately delineated the horizons on our view from the CPU — in eight-bit processors we are confined, except in particular circumstances, to