More words of wisdom from everyones favourite aunt

It is almost inevitable that at some stage the competent Basic programmer will turn his attention to machine code. The promise of speed, compactness of code and even the prospect of designing custom EPROMs soon becomes too much to ignore and the first tentative steps in assembly language programming are taken.

That is a very delicate time, because a bad experience with an early program can soon deter the beginner from machine code for life. Having a good book which lists many examples helps but in the long term a good assembler will save a good deal of bloodshed.

What makes on assembler better than another? What is an assembler anyway? Is machine code not just machine code?

Machine code is the language microprocessors speak and it was never designed for humans to use. It consists of numbers, each representing a simple function, such as 'Add two values'. Each microprocessor on the market uses a different machine code, which makes things even more difficult. At any large computer club you will find Z-80, 6502 and 68000 enthusiasts, all arguing that their particular processor is the best and all having good reasons for thinking so.

Although the exact details of programming varies from processor to processor, the principles remain the same and, once thay are learned, moving to a different system is a relatively painless process.

To make life easier, various translation programs have been written in machine code. One of the most popular is Basic, which was designed in the 1960s to allow beginners to use computers. Because each Basic statement must be converted to machine code before it can be executed, speed is not at a premium.

Apart from several notable exceptions, all home computers have a Basic program, usually on ROM and available from switch-on. A 'standard' for Basic is impossible, because each new computer available has new features, each requiring new commands to control them. A case in point is the Locomotive Software Basic 'sound' command. Trying to get this command to work on a Spectrum or a Commodore is hopeless.

John Kennedy leaps into action with solutions galore. This month machine code, the ins and outs; assemblers, translators to befriend and understand and the importance of logical thought and extreme patience.

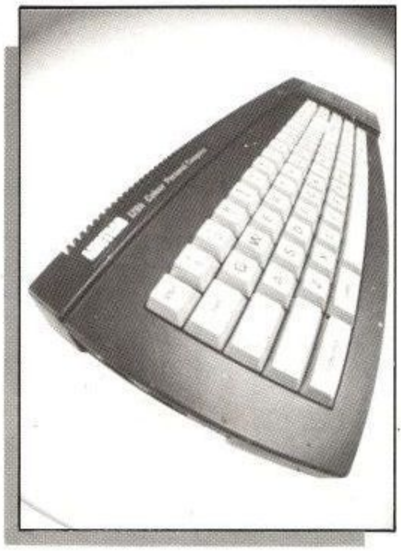
An alternative to Basic and other high-level languages including Pascal, C and possibly Forth, is assembly language, in which case each machine code function is represented by a single, almost-English world called a mnemonic. Thus with a Z-80 processor, like the one in CPCs, to add 42 to the contents of the internal accumulator register, instead of writing 'C6 2A', you would write "ADD A,42".

The program which translates assembly language mnemonics to machine code is called an assembler. For the Amstrad range of micros alone there are many assemblers available, each with its own little quirks. Of course, you still need to know what is an internal accumulator register and for that you should consult one of the many books or magazine articles available. It cannot be emphasised sufficiently that the beginner should acquire an assembler if he is in any way serious about learning machine code.

Programs written in machine code are faster and more compact than their Basic counterparts. All commercial games are written in code. Operating systems, compilers, assemblers and word processors are written in code. Any program in which speed is important, or in which large amounts of memory are involved, must be written in code to be efficient.

An assembler converts a list of mnemonics, called the SOURCE file, into machine code, the OBJECT file. With an operating system such as CP/M, the source file is a standard ASCII file and the object code is another disc file in a different format, usually either *.HEX or *.REL. Those object files must be put through another program to produce executable machine codes. The programs ASM.COM and LOAD.COM are supplied on the CP/M system discs with the CPC disc computers for this purpose.

With simpler systems the source file is usually typed into an editor program and, with it still in memory, the assembler is evoked. The resultant object code is sent directly to another section of memory where it may be executed or saved to tape or disc. As this technique involves four files in memory at once – the editor program, the assembler program, the source file and the object data – things become rather squashed when working with larger programs.


Old faithful

Putting the editor and assembler programs on ROM eases things, as does using the disc or tape to store the source code and reading it in a line at a time for assembly.

Apart from the obvious advantages of not needing to remember all the machine code numbers and their related functions, an assembler helps in other ways. One of the most useful features is the concept of 'labels'. Imagine that you needed to call a routine at hex address & BD5A several times in your program. Think how much better it would be if you could refer to it as a simple English name, such as PRINT A CHAR instead of the hex number.

An assembler will do this for you. It will also calculate jump addresses, so you do not have to worry about counting the number of bytes your 'JR' – jump relative – instruction uses. The list of these labels and their linked hex values can all be listed after assembly if you wish.

Assemblers will also make saving and loading files much easier and some will even make sure your object code is not over-writing something important accidentally.

A 'monitor' is a special program which is sometimes incorporated into the assembler but is strictly a program in its own right. It provides various house-keeping facilities, such as moving round large blocks of memory or, more usefully, allows the testing of machine code programs by means of breakpoints and single-stepping.

Breakpoints are special mnemonics you put in your source code and, when the program is assembled and run, the contents of all the registers will be displayed. Single-stepping is similar, except that each machine code instruction in turn is executed, with the register contents displayed each time. Monitors are very useful for the beginner, as they show what is happening at any time.

There are many assembler packages available for the CPC range but we concentrate on those not operating under CP/M but AMSDOS. It is unlikely the newcomer would survive a head-on confrontation with CP/M without permanent mental scarring.

All assemblers have a set of pseudo-mnemonics or directives. They are special instructions which, although not Z-80 assembly language, are treated as such by the computer. As there is no real standard set of directives, each assembler uses its own and this can cause problems for the beginner. Table one lists the various directives and their meaning.

Another difference between assemblers is their speed of operation. This may not be apparent for short programs but when you are assembling your 16K game for the umpteenth game, a slow assembler will waste a good deal of time.

A reasonable assembler is one which will make programming in assembly language as easy as possible. Disc-based computers will also make things less troublesome but it is still possible to learn how to program using the simplest of asemblers and a tape recorder. The important skills are logical thought and extreme patience.

Table 1. Assembler directives.

ORG xxxx, ORIGIN xxxx

xxxx is the address where the object code is to be stored. Most asemblers will also allow a second address to be specified. In this case the first address is where the code THINKS its going, and the second address is where it is going. This is useful if you are writing a program where the location of the machine code is important but it cannot be placed there at assembly time for one reason or another – e.g., the assembler or source code already occupies that address.

Examples

ORG &4000

Store the object code from address &4000 onwards.

ORIGIN &4000, &8000

Store the object code at address &8000 onwards but assembled to run at &4000. Obviously to execute it, it must be saved and re-loaded at the correct (&4000) address.

EQU, EQUATE.

Assign an address to a label. A label can replace any 16-bit number. Unfortunately, each assembler has its own likes and dislikes concerning labels. Some allow any character or number, some limit the length to six characters. Labels can also be inserted anywhere in the source code to represent an address.

Examples:

ASCII for Asterisk EQU 42

TXTOUT EQUATE &BB5A

```
      ld b,10
.loop call print-number
      djnz loop
```

BYTE, DB, DEFB.

Store a single byte or an ASCII string in byte form in memory. Several items may be separated by commas.

Examples:

db 10
byte "Hello mother",&ff

WORD, DW, DEFW.

Store a two-byte value in memory.

Example:

dw &c000,&c050,&c0a0

RMEM, DS, DEFS.

'Reserve Memory' or 'Data Storage'. The value following this directive specifies how many bytes are to be reserved. On some assemblers an optional second parameter allows the reserved memory to be set to a certain value.

Examples:

DS 100

Set aside 100 bytes.

RMEM 100,42

Set aside 100 bytes, all with the value 42.

END, STOP

A marker to indicate the end of the source code.

DUMP, LIST

Produce a list of the labels used in the program.

Most assemblers have many more directives, allowing conditional assembly and control of printer output. Note also that 'END' and 'STOP' may mean different things to different programs, as may 'DUMP' and 'LIST'.