GENERAL ROUTINE

Making machine code programs relocatable, so that their execution is independent of their locations, requires the use of symbols and labels rather than absolute addresses and values. We study some more assembler directives and their role in program structure, and take a first look at Assembly language subroutine calls.

Because Assembly language is essentially a simple programming language composed of the 'primitive' commands that the CPU can manage, you will find yourself constantly writing and re-writing fragments of program to do the same essential tasks that you take for granted as part of the instruction set of a high-level language — input/output handling, for example, or two-byte arithmetic routines. The sensible thing to do is to establish a library — on tape, disk or paper — of the most commonly used routines, and merge these into new programs as the need arises.

There are two major problems associated with this, however. The first is the difficulty of writing important, and often lengthy, routines in a sufficiently general way that they can be inserted in different programs without adjustment or re-writing. The second problem is in writing useful routines that are not rooted in one set of memory locations, so that they can be relocated in memory through a new assembly with a different ORG address, and perform exactly the same function there as in their original locations.

Both problems are aspects of the generality/portability problem familiar to BASIC programmers, and are solved in much the same way — by using variables to pass values from program to subroutine; by using local variables in subroutines to make them independent of the larger program context; and by avoiding the use of absolute quantities (both numerical or string constants) and program line numbers.

In Assembly language programming we have become used to the idea of memory locations as the equivalent of BASIC variables — programs operate on the contents of the locations, whatever those contents might be, in the same way that a BASIC program operates on the contents of its variables. Unfortunately, we have tended to refer to memory locations by their absolute addresses, a convenient habit at first, but one that must now be renounced in the name of generality. The answer is to use symbols instead of absolute addresses and values, and to use the range of symbolic forms offered by assembler pseudo-opcodes as the equivalents of both variables and program line numbers. We have seen examples of both uses already. Consider this program, for example:

6502			Z80		
DATA1	EQU	$12	DATA1	EQU	$12
DATA2	EQU	$79	DATA2	EQU	$79
	LDA	DATA1	LD	A,(DATA1)	
LOOP	ADC	DATA2	ADC	A,(DATA2)	
	BNE	LOOP	JR	NZ,LOOP	
	RTS		RET		

Here we have two kinds of symbol, two values and a label, all used as the operands of the Assembly language instructions. Because of this, the program fragment is both general and able to be relocated. The only absolute quantities are the values of DATA1 and DATA2, and they can be initialised in the surrounding program, rather than at the start of the routine itself.

There are other pseudo-ops that we have not yet discussed. In particular, DB, DW and DS (though, like ORG and EQU, they may differ from one assembler program to another). These three directives, which stand for 'Define Byte', 'Define Word', and 'Define Storage', enable us to initialise and allocate memory locations, as in this example:

			ORG	$D3A0
D3A0	5F	LABL1	DB	$5F
D3A1	CE98	LABL2	DW	$98CE
D3B3		LABL3	DS	$10
D3B3		DATA1	EQU	LABL3

SYMBOL TABLE:
LABL1 = D3A0: LABL2 = D3A1: LABL3 = D3A3
DATA1 = D3A3
ASSEMBLY COMPLETE — NO ERRORS

In this full Assembly listing (the output of an assembler program) we see at the bottom for the first time a symbol table, consisting of the symbols defined in the program and the values they represent. There are several important things to notice in this fragment. First of all, in the line that begins LABL1, the DB pseudo-op is used. We can see from the listing that the ORG directive has given the address $D3A0 to LABL1, and the symbol table confirms this. The effect of DB here is to place the value $5F in the byte addressed by LABL1 — so memory location $D3A0 is initialised with the value $5F, as we can see in the machine code column of the listing.

Secondly, LABL2 represents the address $D3A1. However, DW has the effect of initialising a 'word' (two consecutive bytes) of storage, so the value $98CE is stored in locations $D3A1 and $D3A2 in lo-hi form — this can be seen clearly in the machine