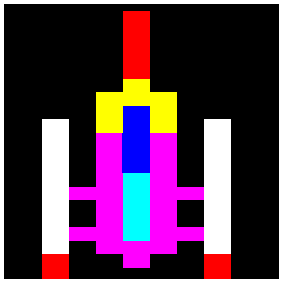
SECTION 2



7

MACROS

If an assembler programmer wants to use a block of instructions several times during a single program then this block only needs to be entered once. There are then two methods of using this block when it is needed. The first is to put a label before it and an RTS instruction at the end, and reference it as a subroutine using the JSR instruction which was described earlier. In this case the CPU will 'jump' to the label when it reaches the JSR and 'jump' back again when it reaches the RTS at the end of the subroutine.

The alternative method is to turn the block of instructions into a macro, the method for doing this will be explained later. Essentially what this does is to give this block of instructions a name and, when the assembler comes across this name, it inserts the instructions of the macro into the object code which it is producing so that the CPU does not have to perform any jumps when it is executing the code.

Hence the main difference between macros and subroutines is that subroutines are called at run-time and macros are called at assembly time.

The advantage of macros is that they are faster than subroutines since no jumps are needed during

execution. The disadvantage is that if the macro is to be used several times, the resulting machine code program will have to contain multiple copies of the instructions represented by the macro, and thus be long. Using a subroutine would only require one copy of the instructions. Macros can be more useful than just an aid to save typing however, and this chapter explains some of their other features. Further examples can be found in section 12.5 (General purpose macros).

7.1 Generating and calling a macro
Consider the sequence of instructions:

```
ROR A : ROR A : ROR A : ROR A
```

This simply shifts the upper nibble of the accumulator into the lower nibble. A macro with the name ' FNrotateacc'containing this sequence can be set up outside the assembler program as follows:

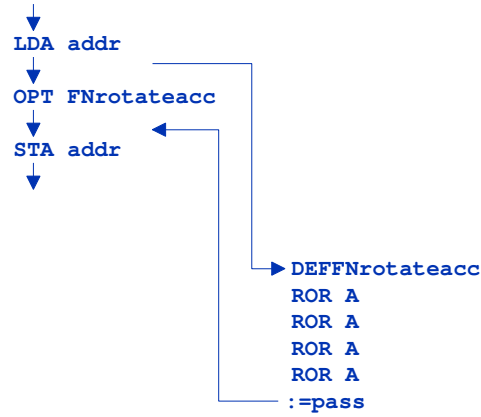
```
 DEF Fnrotateacc
[OPT pass
 ROR A : ROR A ROR A : ROR A
] :=pass
```

This macro can then be called from inside the assembler with the statement

```
OPT FNrotateacc
```

The OPT statement is being used here as a dummy statement, simply to call FNrotateacc and have no other effect. We therefore arrange for the value of the function to be ' pass'which should be the value used in the initial OPT statement. Therefore on reaching this statement the assembler will generate the machine code corresponding to the assembler instructions of the macro, and place this in the object code. Then it will move on to the next instruction of the assembler program.

The flow of control when the program is being typed will look something like this:



The machine code produced will be as follows:

```
A5 81     LDA addr          addr = &81
6A        ROR A
6A        ROR A
6A        ROR A
6A        ROR A
85 81     STA addr
```

7.2 Macro parameters

Macros can take parameters, thus the previous example could be rewritten in such a way that it could rotate the accumulator any number of times (as long as this number is greater than 0):

```
 DEF FNrotateacc(rotate)
 FOR number 1 TO rotate
[OPT pass
  ROR A
]
 NEXT number
 = pass
```

So, to rotate the bits in any memory location any number of times to the right, simply set up a macro as follows:

```
 DEF FNrotate(address, rotate)
 FOR number = 1 TO rotate
[OPT pass
   ROR address
]
 NEXT number
 =pass
```

A typical call might be

```
OPT FNrotate(&3000, 4)
```

This would generate machine code to rotate right four times the bits in location &3000.

7.3 Conditional assembly in macros

Macros can also be constructed to contain conditional instructions, so that they will assemble different pieces of code according to the parameters passed. For example, the following macro works out the shortest way of rotating the accumulator left:

```
DEF FNoptimumrotate(rotate)
IF rotate < 1 THEN = pass
IF rotate < 5 THEN FOR number = 1 TO rotate :
                  [OPT pass : ROL A:]:Next number
                  ELSE FOR number = 1 TO (9 - rotate) :
                  [OPT pass : ROR A:]:Next number

= pass
```

7.4 Labels in macros

Labels cannot be used in the normal way inside macros. Consider, for example, the macro given below:

```
 DEFFNstar
[OPT pass
   LDX addr
   BEQ exclamation
   LDA #ASC"*"
   JSR oswrch
.exclamation
   LDA #ASC"!"
   JSR oswrch
] : =pass
```

When the assembler reaches the ' BEQexclamation' loop instruction on the second pass, it will give the offset the same address of the label which was set up the first time around. If forward referencing is not used then this problem will not occur, but it is still undesirable to use label names time and again. The program becomes very difficult to follow and to debug.

To use labels in macros that are called from more than one place, it is necessary to set up tables of labels. Thus if the label ' start'is used in a macro, an array called ' start'would have to be DIMensioned at the beginning of the program together with as many elements as the number of times the macro is called. If the macro with ' start'in it was called three times from within the program, the statement ' DIM start(2)' would have to be inserted before the first call to that macro took place. Also, each call to the macro would have to pass a parameter which contained a number (0,1 or 2) so that the correct label would be used. To illustrate:

```
 DEF FNmacro(fred,jim,no)
[OPT pass
  LDA fred
  LDY jim
.start(no)
  JSR oswrch
  DEY
  BNE start(no)
]
=pass
```

The above macro will print the contents of ' fred'as an ASCII character, ' jim'times. By convention the label number is always passed as the last parameter, and it is also a good idea to have all the macros using the same variable to hold the number (in this case ' no').

A typical call to the above macro might be

```
OPT FNmacro(addr, 45, 1)
```

This would use the label ' start(1)' .