5 Macros

When writing a program in assembly language, you frequently encounter situations where a group of instructions is repeated throughout your program with only minor variations. Subroutine calls which require parameters to be setup are a typical example. In this case, only the arguments are different. What is needed is a way to define a template of instructions which could then be invoked at those points in the program where they are needed. *Macros* are exactly these templates.

5.1 General Operation

Before a macro can be used it must first be defined. This definition *must* be processed by the assembler on source lines that are read before the source lines on which the macro is used. This definition includes the *name* of the macro, a *body* which includes the fixed elements as well as where parameters are to be substituted and finally an *ending* which tells the assembler that the definition is complete.

Once the definition is completed the macro may be used. The *name* of the macro used in the *opcode* field of a subsequent source line is what actually invokes the macro. The substitution parameters are placed in the *operand* field separated with commas. The previous macro definition now is included at this point in the program very much like a copy. The main difference is that the parameters included on the invocation line are substituted throughout the source lines as defined in the template. These lines are then assembled and optionally listed.

5.2 Macro Definition

A macro is defined with a *MACRO* directive. The operand field *must* contain a macro name of up to 6 characters. Note that the assembler can distinguish between an identifier and a macro with the same name. However, the assembler *cannot* distinguish between a macro and a predefined assembler opcode or directive of the same name. Following the *MACRO* directive is a number of source lines that constitute the template. The definition ends with an *ENDM* directive which has no operand field. Up to 30 macros may be defined whose templates use no more than a total of 1.5K bytes of memory.

Substitution parameters are indicated in the template lines with the percent sign (%) followed by single digit number (0 through 9). Up to

9 parameters numbered from 1 to 9 can be used. The zeroth parameter is a macro expansion count which is automatically kept by the assembler. Use of this parameter can guarantee a unique value for each expansion of the macro. This is useful when including LABEL fields in the template. Example:

```
          MACRO PASFUN
          LDD   STATICBASE,PCR
          LEAY  %1,PCR
          LEAX  %2,PCR
          PSHS  U,Y,X,D
          LBSR  PASFUN
          LEAS  6,S
          LDD   ,S++
          BEQ   PASFUN%0
          ADDD  #0%3
PASFUN%0  EQU   *
          ENDM
```

The above example generates a postion-independent call sequence to a Pascal function. The function requires two parameters whose addresses are loaded into the Y and X registers respectively. The D register is always loaded with a given base value. The PSHS sets up the stack with the U register push used only to reserve space for the value returned by the function.

On return, the macro cleans up the stack and gets the returned value in the D register via a LDD rather than a PULS in order to set the CC register. A check then follows which adds a third macro parameter to the result if a non-zero was returned by the function. Note that the third macro parameter is optional in that if it is not present, a zero is added to the result. This macro also makes use of the macro expansion counter to create an identifer for the macro's own use.

Note that the macro definition itself does not result in *any* code generation. Neither does the assembler try to parse the template so assembly errors may occur when the macro is invoked.

5.3 Macro Invocation

A macro is invoked by using its name in the *opcode* field of a source line that follows the definition. Up to 9 parameters may be included in the *operand* field separated with commas. Not all parameters need be included in a given invocation. All parameters following the last one specified, as well as those that are explicitly not included via placeholder commas, are assigned a null value.

The previous macro defintion is then included at this point in the program. The parameters included (or not included as the case may be) on the invocation line are substituted throughout the source lines as defined in the template. These lines are then assembled and optionally listed.

An example invocation of the macro defined above follows:

```
             PASFUN STRING,COUNT,7
+            LDD    STATICBASE,PCR
+            LEAY   STRING,PCR
+            LEAX   COUNT,PCR
+            PSHS   U,Y,X,D
+            LBSR   PASFUN
+            LEAS   6,S
+            LDD    ,S++
+            BEQ    PASFUN1
+            ADDD   #07
+PASFUN1     EQU    *
```

In the above example, the three arguments are substituted where the %1, %2 and %3 are found in the template. The macro expansion count is 1 and is substituted where %0 is found in the template. The following example shows a second invocation of the same macro:

```
             PASFUN STRING1,COUNT+2
+            LDD    STATICBASE,PCR
+            LEAY   STRING1,PCR
+            LEAX   COUNT+2,PCR
+            PSHS   U,Y,X,D
+            LBSR   PASFUN
+            LEAS   6,S
+            LDD    ,S++
+            BEQ    PASFUN2
+            ADDD   #0
+PASFUN2     EQU    *
```

Notice that the third parameter was not included on the invocation line. Since the macro was constructed with a leading zero before the %3 in the ADDD line, its presence is not required for an error-free assembly. In addition, %0 was substituted with a 2 instead of a 1 providing a unique label for both macro expansions. Note that in this macro a unique label is not absolutely required since the length of the branch is always the same and could be indicated by *+5.

AsmLang

6 Linkage Directives

DEFT Macro/6809 provides directives that allow the object code produced by one assembly to be combined with that of others. The primary uses of this separate assembly facility are:

- A very large program can be divided into managable pieces which are then individually coded and assembled.
- A frequently used routine or set of routines can be written and tested once. Any programs that subsequently need these routines can merely reference them and then include them with the **DEFT Linker.**
- Assembly language programs can be easily combined with PASCAL programs.

When talking about separate assembly the term *module* is used to refer to the code that is assembled via one execution of the assembler. Linking these modules together is accomplished by declaring a given identifier as *public* in the module in which it is defined. Other modules which wish to use the routine or data area defined with this identifier declare it as *external*. **DEFT Linker** then inserts the correct absolute address or offset into the code when the final binary image is created.

6.1 PUBLIC

The *public* directive is used to declare an identifier as public. The identifier must be defined elsewhere in the same module. The *operand* field contains the identifier that is to be declared *public*. Example:

```
        PUBLIC    MYSUBR
           .
           .
           .
MYSUBR  PSHS      Y,X,D   Save Registers
           .
           .
           .
```

As with any other reference to an identifer, a *public* directive can come either before or after the identifier is defined. Note that an identifier which is defined as *ext* or *exta* cannot be given the *public* attribute.

6.2 EXT and EXTA

The *ext* and *exta* directives are used to define an identifier and to declare it as *external* to this module. *Ext* defines a relative identifier. *Exta* defines an absolute identifier. The distinction does not affect the code that is generated by the assembler, but it does allow the assembler to correctly flag PIC and non-PIC code. The identifier to be defined is in the *label* field. Example:

```
YOURSUBR    EXT
YOURCONST   EXTA
             .
             .
             .
            LDD    #YOURCONST
            LBSR   YOURSUBR
```

6.3 STACK

This directive allows you to specify how much stack space this module will require at execution time. This is convenient when linking assembly language with **DEFT Pascal** so that a total stack requirement can be determined by **DEFT Linker**. If this directive is not present, the assembler assumes a zero stack requirement. The absolute expression in the *operand* field is the amount. Example:

```
            STACK  $20
```

Asm Lang

7 Listing Control Directives

This section describes the assembler directives available to control the source listing produced by the assembler. Although these directives control the source listing, they are not included in the listing themselves.

7.1 EJECT

The assembler normally prints 55 source lines on a page before starting a new page. This directive specifies that the next source line should begin at the top of a page. There is no *operand* field. Example:

```
EJECT
```

7.2 LIST

This causes the assembler *list level* to be incremented by one. The *list level* is a value that starts at zero and determines whether source lines should be included in the list file. When this value is greater than or equal to zero, lines are included. When it is negative, source lines are not included. If a previous NOLIST (see below) made the *list level* go negative, then this directive will cause the listing to be turned back on. If the *list level* is already zero, this LIST will cancel the next following NOLIST. This directive has no *operand*. Example:

```
LIST
```

7.3 NOLIST

This causes the assembler *list level* to be decremented by one. See *LIST* for a description of the *list level*. Its general purpose is to prevent source lines from being listed. This directive has no *operand*. Example:

```
NOLIST
```

7.4 MLST

This directive causes macro expansions to be listed. Unlike the *LIST* directive, the *MLST* directive does not have a *level*. It is either on or off. This directive has no *operand*. Example:

```
MLST
```

7.5 NOMLST

This directive suppresses macro expansions from being listed. This directive has no *operand*. Example:

```
                NOMLST
```

7.6 SKIP

This directive causes 1 or more blank lines to be included in the source listing. The number of blank lines included is the absolute expression in the *operand* field. If the *operand* field is not present or the expression is less than 1, then a value of 1 is used. Example:

```
                SKIP    2
                SKIP
```

7.7 STITLE

This directive specifies the string that is to be the subtitle string printed at the top of the listing starting with the next page. The string found in the *operand* field is used. This directive does an implicit *EJECT*. Example:

```
        STITLE /Important Subroutine Name/
```

7.8 TITLE

This directive specifies the string that is to be the title string printed at the top of the listing starting with the next page. The string found in the *operand* field is used. This directive does an implicit *EJECT*. Example:

```
        TITLE /Important Program Name/
```

8 Error Messages

The **DEFT Macro/6809** Assembler generates error messages in the source listing at those points where it detects either syntax errors or encounters I/O errors while processing a source file. Error messages are distinguished by the *** *ERROR* - at the beginning of the line and follow the line that they are referencing. Following are the error messages and a short explanation of each.

ADDR MODE

An invalid addressing mode was used.

BAD OPCODE

An unknown opcode or macro was used.

BAD RMB

An RMB instruction must have a positive absolute expression for an *operand.*

COPY NEST

A copied file may not have a COPY instruction in it.

DUPL MACRO

There is already a macro defined with this name.

DUPL SYMBL

There is already a symbol defined with this name.

PUBLC->EXT

An external symbol is being declared as public. This is illegal.

EXPRESSION

An illegal expression has been detected.

LABEL RQ'D

This opcode requires a symbol in the label field and there is none.

MAC SPACE

This macro definition exhausts all the available macro space and so is rejected.

MACRO NEST

You cannot invoke a macro from within a macro.

OPRND RQ'D

This opcode requires an *operand* and there is none.

OPRND SIZE

This opcode requires an 8 bit *operand* and the one that is present requires 16 bits.

PHASE

This label is being assigned a different value on the assembler's second pass than it recieved on the first pass. This is usually due to using a symbol in an RMB statement before the symbol is defined.

REGISTER

An unknown or illegal register has been specified.

UNDEF SYM

An unknown or illegal symbol has been used.