

# Chapter 7

## Macros and Preprocessing Directives

### 7.1 Introduction

The use of macros is intended to facilitate the design and writing of assembly code. Besides the use of sub-procedures, writing macros is another alternative to the organization of source code besides the use of procedures. However, while a *call* to a sub-procedure allows the activation of the corresponding code in memory for execution, macros will expand the source code during development. The name of the macro which occupies one line of code is replaced by the source code during assembly. Macros are similar to an include file. A sequence of text lines is defined and named only once but referred to several times, and each time, the reference is substituted by the same code. The only advantage of calling macros over calling procedures is the fact that no *call* and *ret* instruction is executed. However, the speed gained has to be paid with the cost of the extra memory space required for each expansion. Therefore, while macros provide speed, procedures provide compactness.

### 7.2 Macro Definition

A macro definition makes use of the directives *%macro* and *%endmacro* in the following format:

```
%macro name_of_macro [p]
```

```

    ...
    code
    ...
%endmacro

```

A number  $p$  of parameters will be expanded with the values given as arguments when the macro is invoked. Parameters may be referenced anywhere within the macro and identified with an integer  $i$  preceded by the symbol “%i”, where  $1 \leq i \leq p$ . Thus, the first parameter is identified as %1, the second parameter as %2 and so on until the last parameter numbered  $p$  is identified as % $p$ . A macro invocation uses the name of the macro followed by the list of arguments as follows:

```
name_of_macro [arguments]
```

Parameters are replaced from the argument list from left to right. If the number of arguments is less than the number of parameters, the instructions that refer to them will not be encoded and an assembly error is generated. On the other hand if the number of arguments is greater than the number of parameters, then the extra arguments from left-to-right will be ignored.

*Example:* This macro will be called to display a character using *int 21*.

```

%macro putch 1
    mov dl, %1
    mov ah, 02h
    int 21h
%endmacro

```

One use of the above macro is to produce a beep sound with the following line:

```
putch '7'
```

*Example:* Write a macro to display a \$-terminated string.

```

%macro display 0
    push ax
    mov ah, 9
    int 21h
    pop ax
%endmacro

```

Note the use of the *push* and *pop* instructions to save and restore the contents of *ax* using the stack. As is the case for sub-procedures, for complex macros saving and restoring registers safeguards the integrity of the overall program and provides coding flexibility as most of the *cpu* resources are available for writing macros.

The use of *display* requires a pointer to the string and must be pre-loaded into *dx* before the call is made. If *msg* is such a pointer, then the following sequence will display it:

```
mov dx, msg
display
```

Alternatively, the string pointer can be easily incorporated into the macro declaration as shown in the following example.

*example:* Modify *display* to pass the string pointer as a parameter:

```
%macro display 1
    push ax
    push dx
    mov ah, 9
    mov dx, %1
    int 21h
    pop dx
    pop ax
%endmacro
```

Macros can be called within macros as shown in the example that follows.

*example:* The following sequence will display a string in a position determined by another macro:

```
%macro display_at 3
    locate %1, %2
    display %3
%endmacro
```

and the following sequence shows the implementation of the macro *locate*:

```
%macro locate 2
    push ax
```

```

        push bx
        push dx
        mov bx, 0           ;page 0
        mov ah, 2           ;code function
        mov dh, %1          ;row
        mov dl, %2          ;column
        int 10h
        pop dx
        pop bx
        pop ax
    %endmacro

```

and a typical invocation of *display\_at* would be as follows:

```
display_at row, column, string
```

The macros defined so far are case sensitive standard macros, referred to as *multi-line* macros. Case insensitive multi-line macros can be defined using the alternative directive *%imacro*.

Compounded values that include a comma, can be passed in the list of arguments by enclosing the entire parameter in braces. For example the following macro:

```

%macro silly 2
    %2: db %1
%endmacro

```

can be used and expanded as follows:

```

silly a, letter_a           ; letter_a: db a
silly ab, string_ab        ; string_ab: db ab
silly {13,10}, crlf        ; crlf: db 13,10

```

## 7.3 Emulation of C calls

Large projects may require that assembly procedures that have been written to be callable from C to be also callable from another assembly procedure. In this case is convenient to simply write a macro that emulates the way in which C calls use the stack to save and restore parameters each time the call is made. A macro to emulate C calls will therefore require the following steps:

- Push parameters into the stack in reverse order in which they appear in a typical C call,
- Execute a call to the procedure,
- Restore the stack pointer

*Example:* The following example illustrates the passing of parameters to a 16-bit procedure named *video* with the following prototype:

```
video{string_name, row, col, attr, length, page}
```

*Video* displays a string in a selected page at the coordinates given by *row* and *col*. The emulation macro requires 6 parameters and will allow any 16-bit *nasm* program to call *\_video* through a call to the macro *video*:

```
%macro video 6
    mov ax, %6
    push ax
    mov ax, %5
    push ax
    mov ax, %4
    push ax
    mov ax, %3
    push ax
    mov ax, %2
    push ax
    mov ax, seg %1    ;push segment part
    push ax
    mov ax, %1        ;push offset part
    push ax
    call far _video
    add sp, 14
%endmacro
```

Note that parameter %1 corresponds to the message string with a *segment:offset* pointer. The segment part is pushed onto the stack using the pseudo-operator *seg*. A second push instruction is needed to save the offset part into the next word size location in the stack. Once all parameter values are saved a far call is made to *\_video* which in turn automatically saves *cs:ip* into the next two word locations in the stack. Recall that this is a pointer to the next instruction (*add sp, 14*) and when

*\_video* executes a return this pointer is restored into *cs:ip*. Now the last instruction simply adjusts *sp* to its value before the call to the macro. The *video* macro can be called from any assembly program as follows:

```
video string_name, row, col, attr, length, page
```

## 7.4 Local Labels

Since all labels must resolve to an address then labels must be unique, i.e., the same label can not be used to reference different addresses. However, this is precisely the problem that results from the expansion of a macro as it will possibly generate multiple instances of the same label. A program with repeated labels will not be assembled. In *nasm* all labels defined within a macro are considered local and handled accordingly by the assembler. A label is made local by preceding it with the sequence: “%%” and inserting a colon after the name. When the assembler recognizes a local label it generates a unique identifier using the same name preceded by the the prefix “.@” and a four-digit number. This number is then incremented each time the same local label is encountered to enforce a unique identifier for each instance.

*Example:* The following macro displays a character every iteration of the loop. The counter is a parameter.

```
%macro repeat 2
    mov cx, %2           ;loop counter
    %%a: mov ah, 2
        mov dl, %1       ;character
        int 21h
        loop %%a
%endmacro
```

## 7.5 Single-line Macros

Single-line macros are defined using the *%define* preprocessor directive. The definitions work in a similar way to C as shown in the following examples:

```
%define cr 10
%define lf 13
%define ctrl 0x1F &
```

```
%define param(a,b) ((a)+(a)*(b))
%define row bp+10
```

The following lines of code and declarations illustrate the invocation of the *defines* above:

```
msg    db    'hello world', cr, lf, '$'
mov byte [param(2,ebx)], ctrl D
mov ax, [row]
```

which will expand to the following code:

```
msg    db    'hello world', 10, 13, '$'
mov byte [(2)+(2)*(ebx)], 0x1F & D
mov ax, [bp+10]
```

Single line macros can be removed with the *%undef* command. For example, the following sequence:

```
%define row bp+10
%undef row

mov eax, row
```

will expand to the instruction *mov eax, row*, since after *%undef* the macro *row* is no longer defined. Macros can also be undefined on the command line using the “-u” option on the *nasm* command line.

Macros defined with *%define* are case sensitive. By using the directive *%ifdef* all case variants are supported.

## 7.6 Conditional Assembly

Conditional assembly directives allow sections of a source file to be assembled only if the stated condition is met. The general syntax of this feature looks like this:

```
%if<condition1>
    code expanded only if <condition1> is met
%elif<condition2>
    code expanded only if <condition2>
```

```

%else
    code expanded if neither <condition1> nor <condition2>
    is met
%endif

```

The clause *%else* is optional, as is the *%elif* clause. More than one *% elif* clause can be used. The following set of directives support general assembly programming:

**%if *expr*:** The conditional-assembly construct *%if expr* will cause the subsequent code to be assembled if and only if the value of the numeric expression *expr* is non-zero. The clause *%if* extends the normal *nasm* expression syntax, to include a set of relational operators such as *=*, *<*, *>*, *≤*, *≥* and *<>* that will test for equality, less-than, greater-than, less-or-equal, greater-or-equal, and not-equal, respectively. The C-like forms *==* and *!=* are supported as alternative forms of *=* and *<>*. Low-priority logical operators such as *&&*, *^^* and *||* are supported, to test for logical *and*, logical *xor*, and logical *or* conditions. These conditions are similar to the C logical operators (although C has no logical *xor*), in that they always return either 0 or 1, and treat any non-zero input as 1 (so that *^^*, for example, returns 1 if exactly one of its inputs is zero, and 0 otherwise). The relational operators also return 1 for true and 0 for false.

**%ifdef:** Beginning a conditional-assembly block with the line *%ifdef macro* will assemble the subsequent code if, and only if, a single-line macro called *macro* is defined. If not, then the *%elif* and *%else* blocks (if any) will be processed instead. For example, when debugging a program, you might want to write code such as:

```

; perform some function
%ifdef DEBUG
    writefile 2,"Function performed successfully",13,10
%endif
; go and do something else

```

Then you could use the command-line option *-dDEBUG* to create a version of the program which produced debugging messages, and remove the option to generate the final release version of the program.

**%ifmacro:** This construct tests for multi-line macro existence. The *%ifmacro* directive operates in the same way as the *%ifdef*. For example, in working with a large project it is convenient to have control over the macros in a library by creating a macro with one name if it doesn't already exist, and another



name if one with that name does exist. The *%ifmacro* is considered true if defining a macro with the given name and number of arguments would cause a definitions conflict. For example:

```
%ifmacro MyMacro
    %error "MyMacro" causes a conflict with an
        existing macro.
%else
    %macro MyMacro
        ; insert code to define the macro
    %endmacro
%endif
```

This will create the macro “MyMacro” if no macro already exists which would conflict with it, and emits a warning if there would be a definition conflict. The *%ifnmacro* construct tests for the macro non-existence. Additional tests can be performed in *%elif* blocks by using *%elifmacro* and *%elifnmacro*.

The following set of directives are provided to support programming using macros:

**%ifidn, %ifidni:** The construct

```
%ifidn text1,text2
```

will cause the subsequent code to be assembled if and only if text1 and text2, after expanding single-line macros, are identical pieces of text. Differences in white space are not counted. *%ifidni* is similar to *%ifidn*, but is case-insensitive. For example, the following macro pushes a register or number on the stack, and allows you to treat *ip* as a real register:

```
%macro pushparam 1
    %ifidni %1,ip
        call %%label
    %%label:
    %else
        push %1
    %endif
%endmacro
```

Like most other `%if` constructs, `%ifidn` has a counterpart `%elifidn`, and negative forms `%ifnidn` and `%elifnidn`. Similarly, `%ifidni` has counterparts `%elifidni`, `%ifnidni` and `%elifnidni`.

**`%ifid`, `%ifnum`, `%ifstr`:** Some macros will want to perform different tasks depending on whether they are passed a number, a string, or an identifier. For example, a string output macro might want to be able to cope with being passed either a string constant or a pointer to an existing string. The conditional assembly construct `%ifid`, taking one parameter (which may be blank), assembles the subsequent code if and only if the first token in the parameter exists and is an identifier. `%ifnum` works similarly, but tests for the token being a numeric constant; `%ifstr` tests for it being a string.

### 7.6.1 Including Other Files

Using a very similar syntax to the C preprocessor, *nasm*'s preprocessor lets you include other source files into your code. This is done by the use of the

```
%include "macros.mac"
```

will include the contents of the file *macros.mac* into the source file containing the `%include` directive. Include files are searched for in the current directory (the directory you're in when you run *nasm*, as opposed to the location of the *nasm* executable or the location of the source file), plus any directories specified on the *nasm* command line using the `-i` option. The standard C idiom for preventing a file being included more than once is just as applicable in *nasm*: if the file *macros.mac* has the form:

```
%ifndef MACROS_MAC
    %define MACROS_MAC
    ; now define some macros
%endif
```

then including the file more than once will not cause errors, because the second time the file is included nothing will happen because the macro *MACROS\_MAC* will already be defined. You can force a file to be included even if there is no `%include` directive that explicitly includes it, by using the `-p` option on the *nasm* command line.

## 7.7 Structure Data Types

The macros *struc* and *endstruc* are used to define a structure data type. The name of the data type is the only parameter in *struc*. In addition to the structure declaration the *struc* macro defines the name of the structure as a symbol with the value zero, and it attaches a suffix *\_size* to specify through an *equ* statement the size of the structure. The fields of the structure are defined with the size of data allocated within the structure. The structure definition is closed with the *endstruc* declaration.

Consider for example the following structure declaration defined with records of students:

```
struc st_record
    name    resb 20
    id      resb 1
    credits resw 1
    status  resb 1
endstruc
```

Any field resolves to a constant displacement with respect to the address assigned to the structure name. For example to access the contents of the field *credit* involves relative addressing with respect to *asmclass* as in *mov ax, [asmclass + credit]*. Alternatively, structures can be declared with a “period” format as follows:

```
struc st_record
    .name    resb 20
    .id      resb 1
    .credits resw 1
    .status  resb 1
endstruc
```

With this format offsets to the structure fields can be defined as in *asmclass.credit*

Once a structure is defined, instances of that structure type can be initialized within the data segment using the *istruc* and *iend* mechanism. For example to declare a structure of type *st\_record* the following instantiation can be used:

```
segment .data
asmclass:
    istruc st_record
        at name,    db  'Jesse Brumbaugh', 0
        at id,      db  '1234567890', 0
```

```

        at credit, dw 100
        at status, db 'senior', 0
    iend

```

The macro *at* makes use of the *times* prefix to advance the assembly position to the correct offset of the specified structure field. Therefore, the structure fields must be declared in the same order as they were specified in the structure definition.

*Example:* the following code illustrates the use of structures, the use of macros to display message in a 32-bit flat mode environment, and the use of *dpmi* services to invoke access to real-mode segments from assembly programs.

```

; Assemble using the 32-bit nasm assembler: nasm32 -f coff v32_ex.asm
; to link under djgpp: gcc -o v32_ex v32_ex_mac.o video32.o

```

```

#include 'dvideo_a.h'      ;provides equates for color attributes

```

```

%macro video 6
    mov eax, %6
    push eax
    mov eax, %5
    push eax
    mov eax, %4
    push eax
    mov eax, %3
    push eax
    mov eax, %2
    push eax
    mov eax, %1
    push eax
    call _video
    add esp,24

```

```

%endmacro

```

```

struc ftest

```

```

    msg:      resb          30

```

```

endstruc

```

```

segment .text

```

```

        global _main
        extern _video

_main:
        mov ax, 0002h                ;dpmi call to
        mov bx, 0b800h
        int 31h                      ;convert B800h to a descriptor format

        mov word [_text_buffer], ax
        video mytest+msg,10,15,LIGHT+WHITE+BLUE_BKG,80,0

        ret

segment .data
        mytest:
            istruc ftest
                at msg,            db        "Hello, World !", 0
            iend

segment .bss
        global _text_buffer
        _text_buffer    resw    1

```

Refer to example 6 in chapter 5 that uses a C-function to request a *dpmi* service for the conversion of the video segment to a descriptor format. In this example the same service is requested using *int 31h*. This example also illustrates the use of a macro to call a display function *video* implemented in 32 bits. Note the differences with the macro used to call a 16-bit implementation of the same function. Since the message is embedded within an assembly structure, the correct offset must be provided to the *video macro* in terms of an instantiation of the *ftest* structure.

### 7.7.1 Array of Structures

Another alternative to initialize structure type data other than the use of *istruc* involves directly accessing structure instances as needed. A large array of structures can be initialized to the same values in each field, or specified fields can be initialized to the desired data. Consider the structure used in the previous example to initialize and display three structure instances for three different messages:

```
; assemble using the 32-bit nasm assembler: nasm32 -f coff strtest.asm
```

```
; to link under djgpp do: gcc -o x strttest.o
```

```
struc strttest
    .msgs resb 30
endstruc
```

```
SEGMENT .text
    global _main
    extern _printf
_main:

    ;initialization

    mov edi, strttest.msgs
    mov dword [edi+0*strttest_size], msg1
    mov dword [edi+1*strttest_size], msg2
    mov dword [edi+2*srtttest_size], msg3

    ;display

    mov edi, strttest.msgs
    push dword [edi+0*strttest_size]
    call _printf
    add esp,4
    push dword [edi+1*strttest_size]
    call _printf
    add esp,4
    push dword [edi+2*strttest_size]
    call _printf
    add esp,4
    ret
```

```
SEGMENT .data
    msg1 db "Hello, World !           ", 10, 0
    msg2 db "Hello, Galaxy !          ", 10, 0
    msg3 db "Hello, Universe !        ", 10, 0
```

Observe that the first instance of the structure *strttest* is initialized with *msg1* at an address given by *strttest.msgs*+0×*strttest\_size*. The second instance is initialized with *msg2* at *strttest.msgs* + 1 × *strttest\_size*, and finally *msg3* initializes the third

instance at  $strtest.msgs + 2 \times strtest\_size$ . Therefore, if  $k$  instances  $istrname$  of a structure type, say  $strname$  are needed, the offset at any field is given by:

$$istrname = strname.n_j + i \times strname\_size$$

where  $i = 1, 2, \dots, k$ , and  $n_j$  denotes any of  $m$  fields  $n_1, n_2, \dots, n_m$  in the structure declaration. The first term provides the offset of the field  $n_j$  within the structure. The second term calculates the offset of the particular instance being accessed. Both the initialization and the display part of the structure *strtest* in the previous example illustrates a straightforward access to its single field *msgs* in each instance.

## 7.8 Exercises