1.3 Installation (Page 21)

1.3.1 Installing NASM under MS−DOS or Windows

Once you've obtained the DOS archive for NASM, nasmXXX.zip (where XXX denotes the version number of NASM contained in the archive), unpack it into its own directory (for example c:\nasm). The archive will contain four executable files: the NASM executable files nasm.exe and nasmw.exe, and the NDISASM executable files ndisasm.exe and ndisasmw.exe. In each case, the file whose name ends in w is a Win32 executable, designed to run under Windows 95 or Windows NT Intel, and the other one is a 16−bit DOS executable. The only file NASM needs to run is its own executable, so copy (at least) one of nasm.exe and nasmw.exe to a directory on your PATH, or alternatively edit autoexec.bat to add the nasm directory to your PATH. (If you're only installing the Win32 version, you may wish to rename it to nasm.exe.) That's it – NASM is installed. You don't need the nasm directory to be present to run NASM (unless you've added it to your PATH), so you can delete it if you need to save

2.1 NASM Command−Line Syntax (p. 23)

To assemble a file, you issue a command of the form

```
nasm −f <format> <filename> [−o <output>]
```

2.1.1 The −o Option: Specifying the Output File Name (p. 23)

NASM will normally choose the name of your output file for you; precisely how it does this is dependent on the object file format. For Microsoft object file formats (obj and win32), it will remove the .asm extension (or whatever extension you like to use – NASM doesn't care) from your source file name and substitute .obj.

2.1.2 The −f Option: Specifying the Output File Format (p. 24)

If you do not supply the −f option to NASM, it will choose an output file format for you itself. In the distribution versions of NASM, the default is always bin; if you've compiled your own copy of NASM, you can redefine OF_DEFAULT at compile time and choose what you want the default to be.

A complete list of the available output file formats can be given by issuing the command nasm −hf.

2.1.3 The −l Option: Generating a Listing File (p. 24)

2.2.1 NASM Is Case−Sensitive (p. 29)

2.2.2 NASM Requires Square Brackets For Memory References (p. 29)

The rule is simply that any access to the contents of a memory location requires square brackets around the address, and any access to the address of a variable doesn't.

2.2.3 NASM Doesn't Store Variable Types (p. 30)

3.1 Layout of a NASM Source Line

```
label: instruction operands ; comment
```

NASM uses backslash (\) as the line continuation character.

3.2.1 DB and friends: Declaring Initialised Data (p. 32)

DB, DW, DD, DQ and **DT** are used to declare initialised data in the output file. They can be invoked in a wide range of ways:

```
db    0x55                ; just the byte 0x55
db    0x55,0x56,0x57      ; three bytes in succession
db    'a',0x55            ; character constants
db    'hello',13,10,'$'   ; string constants
dw    0x1234              ; 0x34 0x12
dw    'a'                 ; 0x61 0x00
dw    'ab'                ; 0x61 0x62 (character constant)
dw    'abc'               ; 0x61 0x62 0x63 0x00 (string)
dd    0x12345678          ; 0x78 0x56 0x34 0x12
dd    1.234567e20         ; floating-point constant
dq    1.234567e20         ; double-precision float
dt    1.234567e20         ; extended-precision float
```

3.2.2 RESB and friends: Declaring Uninitialised Data (p. 32)

RESB, RESW, RESD, RESQ and **REST** are designed to be used in the BSS section of a module: they declare *uninitialised* storage space. Each takes a single operand, which is the number of bytes, words, doublewords or whatever to reserve.

```
buffer:     resb    64          ; reserve 64 bytes
wordvar:    resw    1           ; reserve a word
realarray   resq    10          ; array of ten reals
```

3.2.4 EQU: Defining Constants (p. 33)

The action of **EQU** is to define the given label name to the value of its (only) operand. This definition is absolute, and cannot change later. So, for example,

```
message db 'hello, world'
msglen equ $-message
$ = current location ⇒
$-message = current location - location of start of "message"
          = length of string
```

3.2.5 TIMES: Repeating Instructions or Data (p. 33)

The **TIMES** prefix causes the instruction to be assembled multiple times.

```
zerobuf: times 64 db 0
```

TIMES can be applied to ordinary instructions, so you can code trivial unrolled loops in it: `times 100 movsb`

3.3 Effective Addresses (p. 33)

Effective addresses have a very simple syntax: an expression evaluating to the desired address, enclosed in square brackets.

```
wordvar dw 123
mov ax,[wordvar]
mov ax,[wordvar+1]
```

3.4.1 Numeric Constants (p. 34)

A numeric constant is simply a number.

Suffix: **H** or **0x** for HEX, **Q** or **O** for OCTAL, and **B** for BINARY

```
mov     ax,100          ; decimal
mov     ax,0a2h         ; hex
mov     ax,0xa2         ; hex
```

```
mov     ax,777q             ; octal
mov     ax,777o             ; octal
mov     ax,10010011b        ; binary
```

3.4.2 Character Constants (p. 35)

A character constant with more than one character will be arranged with little-endian order in mind: if you code

```
mov eax,'abcd'
```

then the constant generated is not `0x61626364`, but `0x64636261`.

3.4.3 String Constants (p. 35)

The following are equivalent:

```
db 'hello' ; string constant
db 'h','e','l','l','o' ; equivalent character constants
```

3.5 Expressions (p. 36)

Expressions in NASM are similar in syntax to those in C.

NASM supports two special tokens in expressions

`$` evaluates to the assembly position at the beginning of the line containing the expression; so you can code an infinite loop using `JMP $`.

`$$` evaluates to the beginning of the current section; so you can tell how far into the section you are by using `($-$$)`.

The arithmetic operators provided by NASM are listed here, in increasing order of precedence.

3.5.1 |: Bitwise OR Operator (p. 36)

3.5.2 ^: Bitwise XOR Operator (p. 36)

3.5.3 &: Bitwise AND Operator (p. 36)

3.5.4 << and >>: Bit Shift Operators (p. 36)

3.5.5 + and −: Addition and Subtraction Operators (p. 36)

3.5.6 *, /, //, % and %%: Multiplication and Division (p. 36)

3.5.7 Unary Operators: +, −, ~ and SEG (p. 36)

3.6 SEG and WRT (p. 37)

The `SEG` operator returns the *preferred* segment base of a symbol, defined as the segment base relative to which the offset of the symbol makes sense. So the code

```
mov   ax,seg symbol
mov   es,ax
mov   bx,symbol
```

will load `ES:BX` with a valid pointer to the symbol `symbol`.

`WRT` (With Reference To) keyword.

```
mov ax,weird_seg ; weird_seg is a segment base
mov es,ax
mov bx,symbol wrt weird_seg
```

loads `ES:BX` with a pointer to the symbol `symbol`.

4.1 Single−Line Macros (p. 41)

4.1.1 The Normal Way: %define

Single−line macros are defined using the `%define` preprocessor directive. The definitions work in a similar way to C; so you can do things like

```
%define ctrl 0x1F &
%define param(a,b) ((a)+(a)*(b))

                    mov  byte [param(2,ebx)], ctrl 'D'
```

which will expand to

```
                    mov  byte [(2)+(2)*(ebx)], 0x1F & 'D'
```

When the expansion of a single−line macro contains tokens which invoke another macro, the expansion is performed at invocation time, not at definition time. Thus the code

```
%define a(x) 1+b(x)
%define b(x) 2*x
mov ax,a(8)
```

will evaluate in the expected way to `mov ax,1+2*8`, even though the macro `b` wasn't defined at the time of definition of `a`.

4.1.2 Enhancing %define: %xdefine (p. 42)

To have a reference to an embedded single−line macro resolved at the time that it is embedded, as opposed to when the calling macro is expanded, you need a different mechanism to the one offered by `%define`. The solution is to use `%xdefine`.

4.1.4 Undefining macros: %undef (p. 43)

4.2.1 String Length: %strlen (p. 44)

The `%strlen` macro creates (or redefines) a numeric value to a macro, the numeric value is the length of a string.

```
%strlen charcnt 'my string'
```

In this example, `charcnt` would receive the value `8`

```
%define sometext 'my string'
%strlen charcnt sometext
```

This would result in `charcnt` being assigned the value of `8`.

4.2.2 Sub−strings: %substr (p. 44)

Individual letters in strings can be extracted using `%substr`. An example of its use is probably more useful than the description:

```
%substr mychar 'xyz' 1
%substr mychar 'xyz' 2
; equivalent to %define mychar 'x'
; equivalent to %define mychar 'y'
```

4.3 Multi−Line Macros: %macro (p. 44)

A multi−line macro definition in NASM looks something like this.

```
%macro prologue 1
    push ebp
    mov ebp,esp
    sub esp,%1
%endmacro
```

This defines a C−like function prologue as a macro: so you would invoke the macro with a call such as

```
myfunc: prologue 12
```

which would expand to the three lines of code

```
myfunc: push ebp
mov ebp,esp
sub esp,12
```

The number `1` after the macro name in the `%macro` line defines the number of parameters the macro `prologue` expects to receive. The use of `%1` inside the macro definition refers to the first parameter to the macro call. With a macro taking more than one parameter, subsequent parameters would be referred to as `%2`, `%3` and so on.

4.3.2 Macro−Local Labels (p. 46)

NASM allows you to define labels within a multi−line macro definition in such a way as to make them local to the macro call: so calling the same macro multiple times will use a different label each time. You do this by prefixing `%%` to the label name. So you can invent an instruction which executes a `RET` if the `Z` flag is set by doing this:

```
%macro retz 0
    jnz %%skip
    ret
    %%skip:
%endmacro
```

4.3.3 Greedy Macro Parameters (p. 46)

NASM allows you to define the last parameter of a macro to be *greedy*, meaning that if you invoke the macro with more parameters than it expects, all the spare parameters get lumped into the last defined one along with the separating commas. So if you code:

```
%macro writefile 2+
    jmp %%endstr
    %%str: db %2
    %%endstr:
    mov dx,%%str
    mov cx,%%endstr−%%str
    mov bx,%1
    mov ah,0x40
    int 0x21
%endmacro
```

then the example call to `writefile` above will work as expected: the text before the first comma, `[filehandle]`, is used as the first macro parameter and expanded when `%1` is referred to, and all the subsequent text is lumped into `%2` and placed after the `db`.

4.3.5 %0: Macro Parameter Counter (p. 48)

4.3.6 %rotate: Rotating Macro Parameters (p. 48)

4.6 Including Other Files (p. 54)

Using, once again, a very similar syntax to the C preprocessor, NASM's preprocessor lets you include other source files into your code.

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

will include the contents of the file `macros.mac` into the source file containing the `%include` directive.

4.8.5 STRUC and ENDSTRUC: Declaring Structure Data Types (p. 59)

The macros `STRUC` and `ENDSTRUC` are used to define a structure data type.

`STRUC` takes one parameter, which is the name of the data type. This name is defined as a symbol with the value zero, and also has the suffix `_size` appended to it and is then defined as an `EQU` giving the size of the structure. Once `STRUC` has been issued, you are defining the structure, and should define fields using the `RESB` family of pseudo−instructions, and then invoke `ENDSTRUC` to finish the definition.

For example, to define a structure called `mytype` containing a longword, a word, a byte and a string of bytes, you might code

```
struc          mytype

    mt_long:            resd    1
    mt_word:            resw    1
    mt_byte:            resb    1
    mt_str:             resb    32
endstruc
```

The above code defines six symbols: `mt_long` as 0 (the offset from the beginning of a `mytype` structure to the longword field), `mt_word` as 4, `mt_byte` as 6, `mt_str` as 7, `mytype_size` as 39, and `mytype` itself as zero.

4.8.6 ISTRUC, AT and IEND: Declaring Instances of Structures (p. 60)

Having defined a structure type, the next thing you typically want to do is to declare instances of that structure in your data segment. NASM provides an easy way to do this in the `ISTRUC` mechanism. To declare a structure of type `mytype` in a program, you code something like this:

```
mystruc:
   istruc mytype
   at mt_long, dd 123456
   at mt_word, dw 1024
   at mt_byte, db 'x'
   at mt_str, db 'hello, world', 13, 10, 0
iend
```

5.1 BITS: Specifying Target Processor Mode (p. 64)

The **BITS** directive specifies whether NASM should generate code designed to run on a processor operating in 16−bit mode, or code designed to run on a processor operating in 32−bit mode. The syntax is **BITS 16** or **BITS 32**.

In most cases, you should not need to use **BITS** explicitly. The **aout, coff, elf** and **win32** object formats, which are designed for use in 32−bit operating systems, all cause NASM to select 32−bit mode by default. The **obj** object format allows you to specify each segment you define as either **USE16** or **USE32**, and NASM will set its operating mode accordingly, so the use of the **BITS** directive is once again unnecessary.

The most likely reason for using the **BITS** directive is to write 32−bit code in a flat binary file; this is because the **bin** output format defaults to 16−bit mode in anticipation of it being used most frequently to write DOS **.COM** programs, DOS **.SYS** device drivers and boot loader software.

You do *not* need to specify **BITS 32** merely in order to use 32−bit instructions in a 16−bit DOS program; if you do, the assembler will generate incorrect code because it will be writing code targeted at a 32−bit platform, to be run on a 16−bit one.

When NASM is in **BITS 16** state, instructions which use 32−bit data are prefixed with an 0x66 byte, and those referring to 32−bit addresses have an 0x67 prefix. In **BITS 32** state, the reverse is true: 32−bit instructions require no prefixes, whereas instructions using 16−bit data need an 0x66 and those working on 16−bit addresses need an 0x67.

5.1.1 USE16 & USE32: Aliases for BITS (p. 64)

The **'USE16'** and **'USE32'** directives can be used in place of '**BITS 16**' and '**BITS 32**', for compatibility with other assemblers.

5.2 SECTION or SEGMENT: Changing and Defining Sections (p. 64)

The **SECTION** directive (**SEGMENT** is an exactly equivalent synonym) changes which section of the output file the code you write will be assembled into. In some object file formats, the number and names of sections are fixed; in others, the user may make up as many as they wish. Hence **SECTION** may sometimes give an error message, or may define a new section, if you try to switch to a section that does not (yet) exist.

The Unix object formats, and the **bin** object format (but see section 6.1.3, all support the standardised section names **.text, .data** and **.bss** for the code, data and uninitialised−data sections. The **obj** format, by contrast, does not recognise these section names as being special, and indeed will strip off the leading period of any section name that has one.

5.4 EXTERN: Importing Symbols from Other Modules (p. 66)

EXTERN is similar to C keyword **extern**: it is used to declare a symbol which is not defined anywhere in the module being assembled, but is assumed to be defined in some other module and needs to be referred to by this one. Not every object−file format can support external variables: the **bin** format cannot.

The **EXTERN** directive takes as many arguments as you like. Each argument is the name of a symbol:

```
extern _printf
extern _sscanf,_fscanf
```

Some object−file formats provide extra features to the **EXTERN** directive. In all cases, the extra features are used by suffixing a colon to the symbol name followed by object−format specific text. For example, the **obj** format allows you to declare that the default segment base of an external should be the group **dgroup** by means of the directive

```
extern _variable:wrt dgroup
```

5.5 GLOBAL: Exporting Symbols to Other Modules (p. 67)

GLOBAL is the other end of **EXTERN**: if one module declares a symbol as **EXTERN** and refers to it, then in order to prevent linker errors, some other module must actually *define* the symbol and declare it as **GLOBAL**. Some assemblers use the name **PUBLIC** for this purpose.

The **GLOBAL** directive applying to a symbol must appear *before* the definition of the symbol.

GLOBAL uses the same syntax as **EXTERN**, except that it must refer to symbols which *are* defined in the same module as the **GLOBAL** directive. For example:

```
global _main
_main:
; some code
```

GLOBAL, like **EXTERN**, allows object formats to define private extensions by means of a colon. The elf object format, for example, lets you specify whether global data items are functions or data:

```
global hashlookup:function, hashtable:data
```

5.6 COMMON: Defining Common Data Areas (p. 67)

The **COMMON** directive is used to declare *common variables*. A common variable is much like a global variable declared in the uninitialised data section, so that

```
common intvar 4
```

is similar in function to

```
global intvar
section .bss
intvar resd 1
```

The difference is that if more than one module defines the same common variable, then at link time those variables will be *merged*, and references to **intvar** in all modules will point at the same piece of memory.

Like **GLOBAL** and **EXTERN**, **COMMON** supports object−format specific extensions. For example, the **obj** format allows common variables to be NEAR or FAR, and the **elf** format allows you to specify the alignment requirements of a common variable:

```
common commvar 4:near ; works in OBJ
common intarray 100:4 ; works in ELF: 4 byte aligned
```

Once again, like **EXTERN** and **GLOBAL**, the primitive form of **COMMON** differs from the user−level form only in that it can take only one argument at a time.

Chapter 6: Output Formats (p. 69)

6.1 bin: Flat−Form Binary Output (p. 69)

The **bin** format does not produce object files: it generates nothing in the output file except the code you wrote. Such 'pure binary' files are used by MS−DOS: **.COM** executables and **.SYS** device drivers are pure binary files. Pure binary output is also useful for operating system and boot loader development.

The **bin** format supports multiple section names.

Using the **bin** format puts NASM by default into 16−bit mode. In order to use **bin** to write 32−bit code such as an OS kernel, you need to explicitly issue the **BITS 32** directive.

bin has no default output file name extension: instead, it leaves your file name as it is once the original extension has been removed. Thus, the default is for NASM to assemble **binprog.asm** into a binary file called **binprog**.

6.1.1 ORG: Binary File Program Origin (p. 69)

The function of the **ORG** directive is to specify the origin address which NASM will assume the program begins at when it is loaded into memory.

For example, the following code will generate the longword **0x00000104**:

```
        org     0x100
        dd      label
label:
```

6.2 obj: Microsoft OMF Object Files (p. 70)

The **obj** file format (NASM calls it **obj** rather than **omf** for historical reasons) is the one produced by MASM and TASM, which is typically fed to 16–bit DOS linkers to produce **.EXE** files. It is also the format used by OS/2.

obj provides a default output file–name extension of **.obj**.

obj is not exclusively a 16–bit format, though: NASM has full support for the 32–bit extensions to the format.

The **obj** format does not define any special segment names: you can call your segments anything you like. Typical names for segments in **obj** format files are **CODE**, **DATA** and **BSS**.

If your source file contains code before specifying an explicit **SEGMENT** directive, then NASM will invent its own segment called **__NASMDEFSEG** for you.

When you define a segment in an **obj** file, NASM defines the segment name as a symbol as well, so that you can access the segment address of the segment. So, for example:

```
segment data
    dvar: dw 1234
segment code
    function:
    mov ax,data ; get segment address of data
    mov ds,ax ; and move it into DS
    inc word [dvar] ; now this reference will work
    ret
```

The **obj** format also enables the use of the **SEG** and **WRT** operators, so that you can write code which does things like

```
extern foo
    mov ax,seg foo ; get preferred segment of foo
    mov ds,ax
    mov ax,data ; a different segment
    mov es,ax
    mov ax,[ds:foo] ; this accesses 'foo'
    mov [es:foo wrt data],bx ; so does this
```

6.2.1 obj Extensions to the SEGMENT Directive (p. 71)

The **obj** output format extends the **SEGMENT** (or **SECTION**) directive to allow you to specify various properties of the segment you are defining. This is done by appending extra qualifiers to the end of the segment–definition line. For example,

```
segment code private align=16
```

defines the segment **code**, but also declares it to be a private segment, and requires that the portion of it described in this code module must be aligned on a 16–byte boundary.

The available qualifiers are:

- **PRIVATE**, **PUBLIC**, **COMMON** and **STACK** specify the combination characteristics of the segment. **PRIVATE** segments do not get combined with any others by the linker; **PUBLIC** and **STACK** segments get concatenated together at link time; and **COMMON** segments all get overlaid on top of each other rather than stuck end–to–end.
- **ALIGN** is used, as shown above, to specify how many low bits of the segment start address must be forced to zero. The alignment value given may be any power of two from 1 to 4096; in reality, the only values supported are 1, 2, 4, 16, 256 and 4096, so if 8 is specified it will be rounded up to 16, and 32, 64 and 128 will all be rounded up to 256, and so on. Note that alignment to 4096–byte boundaries is a PharLap extension to the format and may not be supported by all linkers.
- **CLASS** can be used to specify the segment class; this feature indicates to the linker that segments of the same class should be placed near each other in the output file. The class name can be any word, e.g. **CLASS=CODE**.
- Segments can be declared as **USE16** or **USE32**, which has the effect of recording the choice in the object file and also ensuring that NASM's default assembly mode when assembling in that segment is 16–bit or 32–bit respectively.
- The **obj** file format also allows segments to be declared as having a pre–defined absolute segment address, although no linkers are currently known to make sensible use of this feature; nevertheless, NASM allows you to declare a segment such as **SEGMENT SCREEN ABSOLUTE=0xB800** if you need to. The **ABSOLUTE** and **ALIGN** keywords are mutually exclusive.

NASM's default segment attributes are **PUBLIC**, **ALIGN=1**, no class, no overlay, and **USE16**.

6.2.2 GROUP: Defining Groups of Segments

The **obj** format also allows segments to be grouped, so that a single segment register can be used to refer to all the segments in a group. NASM therefore supplies the **GROUP** directive, whereby you can code

```
segment data
; some data
segment bss
; some uninitialised data
group dgroup data bss
```

which will define a group called **dgroup** to contain the segments **data** and **bss**. Like **SEGMENT**, **GROUP** causes the group name to be defined as a symbol, so that you can refer to a variable **var** in the **data** segment as **var wrt data** or as **var wrt dgroup**, depending on which segment value is currently in your segment register.

If you just refer to **var**, however, and **var** is declared in a segment which is part of a group, then NASM will default to giving you the offset of **var** from the beginning of the *group*, not the *segment*. Therefore **SEG var**, also, will return the group base rather than the segment base.

NASM will allow a segment to be part of more than one group, but will generate a warning if you do this. Variables declared in a segment which is part of more than one group will default to being relative to the first group that was defined to contain the segment.

A group does not have to contain any segments; you can still make **WRT** references to a group which does not contain the variable you are referring to.

6.2.6 ..start: Defining the Program Entry Point

OMF linkers require exactly one of the object files being linked to define the program entry point, where execution will begin when the program is run. If the object file that defines the entry point is assembled using NASM, you specify the entry point by declaring the special symbol **..start** at the point where you wish execution to begin.

Chapter 7: Writing 16−bit Code (DOS, Windows 3/3.1) (p. 81)

7.1.1 Using the obj Format To Generate .EXE Files (p. 81)

This section describes the usual method of generating .EXE files by linking .OBJ files together.

Most 16−bit programming language packages come with a suitable linker.

When linking several **.OBJ** files into a **.EXE** file, you should ensure that exactly one of them has a start point defined (using the **..start** special symbol defined by the **obj** format). If no module defines a start point, the linker will not know what value to give the entry−point field in the output file header; if more than one defines a start point, the linker will not know *which* value to use.

An example of a NASM source file which can be assembled to a **.OBJ** file and linked on its own to a **.EXE** is given here. It demonstrates the basic principles of defining a stack, initialising the segment registers, and declaring a start point. This file is also provided in the **test** subdirectory of the NASM archives, under the name **objexe.asm**.

```
segment code

..start:
                mov   ax,data
                mov   ds,ax
                mov   ax,stack
                mov   ss,ax
                mov   sp,stacktop
```

This initial piece of code sets up **DS** to point to the data segment, and initialises **SS** and **SP** to point to the top of the provided stack. Notice that interrupts are implicitly disabled for one instruction after a move into **SS**, precisely for this situation, so that there's no chance of an interrupt occurring between the loads of **SS** and **SP** and not having a stack to execute on.

Note also that the special symbol **..start** is defined at the beginning of this code, which means that will be the entry point into the resulting executable file.

```
                mov   dx,hello
                mov   ah,9
                int   0x21
```

The above is the main program: load **DS:DX** with a pointer to the greeting message (**hello** is implicitly relative to the segment **data**, which was loaded into **DS** in the setup code, so the full pointer is valid), and call the DOS print−string function.

```
                mov ax,0x4c00
                int 0x21
```

This terminates the program using another DOS system call.

```
segment data
hello: db 'hello, world', 13, 10, '$'
```

The data segment contains the string we want to display.

```
segment stack stack
resb 64
stacktop:
```

The above code declares a stack segment containing 64 bytes of uninitialised stack space, and points **stacktop** at the top of it. The directive segment stack **stack** defines a segment *called* stack, and also of *type* **STACK**. The latter is not necessary to the correct running of the program, but linkers are likely to issue warnings or errors if your program has no segment of type STACK.

The above file, when assembled into a **.OBJ** file, will link on its own to a valid **.EXE** file, which when run will print 'hello, world' and then exit.

7.2 Producing .COM Files (p. 83)

While large DOS programs must be written as **.EXE** files, small ones are often better written as **.COM** files. **.COM** files are pure binary, and therefore most easily produced using the **bin** output format.

7.2.1 Using the bin Format To Generate .COM Files (p. 83)

.COM files expect to be loaded at offset **100h** into their segment (though the segment may change). Execution then begins at **100h**, i.e. right at the start of the program. So to write a **.COM** program, you would create a source file looking like

```
 org 100h
section .text
start:
; put your code here
section .data
; put data items here
section .bss
; put uninitialised data here
```

The **bin** format puts the **.text** section first in the file, so you can declare data or **BSS** items before beginning to write code if you want to and the code will still end up at the front of the file where it belongs.

The **BSS** (uninitialised data) section does not take up space in the **.COM** file itself: instead, addresses of **BSS** items are resolved to point at space beyond the end of the file, on the grounds that this will be free memory when the program is run. Therefore you should not rely on your **BSS** being initialised to all zeros when you run.

To assemble the above program, you should use a command line like

```
nasm myprog.asm −fbin −o myprog.com
```

The **bin** format would produce a file called **myprog** if no explicit output file name were specified, so you have to override it and give the desired file name.

7.3 Producing .SYS Files (p. 84)

MS−DOS device drivers – **.SYS** files – are pure binary files, similar to **.COM** files, except that they start at origin zero rather than **100h**. Therefore, if you are writing a device driver using the **bin** format, you do not need the **ORG** directive, since the default origin for **bin** is zero. Similarly, if you are using **obj**, you do not need the **RESB 100h** at the start of your code segment.

.SYS files start with a header structure, containing pointers to the various routines inside the driver which do the work. This structure should be defined at the start of the code segment, even though it is not actually code.

A.3 Running NDISASM (p. 106)

To disassemble a file, you will typically use a command of the form

```
ndisasm [-b16 | -b32] filename
```

NDISASM can disassemble 16−bit code or 32−bit code equally easily, provided of course that you remember to specify which it is to work with. If no **-b** switch is present, NDISASM works in 16−bit mode by default. The **-u** switch (for USE32) also invokes 32−bit mode.

A.3.2 Code Following Data: Synchronisation

Suppose you are disassembling a file which contains some data which isn't machine code, and *then* contains some machine code. NDISASM will faithfully plough through the data section, producing machine instructions wherever it can (although most of them will look bizarre, and some may have unusual prefixes, e.g. '**FS OR AX,0x240A**'), and generating 'DB' instructions ever so often if it's totally stumped. Then it will reach the code section.

Supposing NDISASM has just finished generating a strange machine instruction from part of the data section, and its file position is now one byte *before* the beginning of the code section. It's entirely possible that another spurious instruction will get generated, starting with the final byte of the data section, and then the correct first instruction in the code section will not be seen because the starting point skipped over it. This isn't really ideal.

To avoid this, you can specify a '**synchronisation**' point, or indeed as many synchronisation points as you like (although NDISASM can only handle 8192 sync points internally). The definition of a sync point is this: NDISASM guarantees to hit sync points exactly during disassembly. If it is thinking about generating an instruction which would cause it to jump over

A.3.4 Other Options

The **-e** option skips a header on the file, by ignoring the first N bytes. This means that the header is *not* counted towards the disassembly offset: if you give **-e10 -o10**, disassembly will start at byte 10 in the file, and this will be given offset 10, not 20.

The **-k** option is provided with two comma−separated numeric arguments, the first of which is an assembly offset and the second is a number of bytes to skip. This *will* count the skipped bytes towards the assembly offset: its use is to suppress disassembly of a data section which wouldn't contain anything you wanted to see anyway.

A.3.1 COM Files: Specifying an Origin

To disassemble a **DOS .COM** file correctly, a disassembler must assume that the first instruction in the file is loaded at address **0x100**, rather than at zero. NDISASM, which assumes by default that any file you give it is loaded at zero, will therefore need to be informed of this.

The **-o** option allows you to declare a different origin for the file you are disassembling. Its argument may be expressed in any of the NASM numeric formats: decimal by default, if it begins with **'$'** or **'0x'** or ends in **'H'** it's **hex**, if it ends in **'Q'** it's **octal**, and if it ends in **'B'** it's **binary**.

Hence, to disassemble a **.COM** file:

```
ndisasm -o100h filename.com
```

will do the trick.

a sync point, it will discard that instruction and output a '`db`' instead. So it *will* start disassembly exactly from the sync point, and so you *will* see all the instructions in your code section.

Sync points are specified using the **-s** option: they are measured in terms of the program origin, not the file position. So if you want to synchronise after 32 bytes of a **.COM** file, you would have to do

```
ndisasm -o100h -s120h file.com
```

rather than

```
ndisasm -o100h -s20h file.com
```

As stated above, you can specify multiple sync markers if you need to, just by repeating the **-s** option.