NASM Installation

Installation notes are given in Chapter 1 of NASM.

It's as simple as unpacking a gzipped tar file in a directory of your choice, e.g. /usr/local/src.

- *cd /usr/local/src*

  (You will have to be root to do this.)

- Then type:

  tar -xzvf nasm-0.98.tar.gz

  Note it creates a subdirectory nasm-X.XX (with X replaced with version and minor revision numbers).

- Then *cd nasm-0.98* and type *.configure*

- Type *make* to compile the source and then *make install* to install binaries into /usr/local/bin and man pages into /usr/local/man/man1

NASM Compilation

- To get command line help, type:

  nasm -h
- To compile into an ELF object file .o, type:

  nasm -f elf myfile.asm
- To create a listing file, type:

  nasm -f elf myfile.asm -l myfile.lst
- To send errors to a file, type:

  nasm -E myfile.err -f elf myfile.asm
- To include other search paths such as /usr/include, type:

  nasm -I/usr/include -f elf myfile.asm
- To include other files in a source file, use:

  %include "myinc.inc"
- To define constants, use either of the equivalent forms:

  -dFOO=100 on the compile command line.

  %define FOO 100 in the source file.

NASM is case-sensitive

NASM Syntax

- In order to refer to the *contents* of a memory location, use square brackets.
- In order to refer to the *address* of a variable, leave them out, e.g.,

```
mov eax, bar          ;Refers to the address of bar
mov eax, [bar]        ;Refers to the contents of bar
```

No need for the OFFSET directive.

- NASM does not support the hybrid syntaxes such as:

```
mov eax,table[ebx]    ;ERROR
mov eax,[table+ebx]   ;O.K.
mov eax,[es:edi]      ;O.K.
```

- NASM does NOT remember variable types:

```
data dw 0             ;Data type defined as double word.
...
mov [data], 2         ;Doesn't work.
mov word [data], 2    ;O.K.
```

NASM Syntax

- NASM does NOT remember variable types

  Therefore, un-typed operations are not supported, e.g.

  LODS, MOVS, STOS, SCAS, CMPS, INS, and OUTS.

  You must use instead:

  LODSB, MOVSW, and SCASD, etc.

- NASM does not support ASSUME.

  It will not keep track of what values you choose to put in your segment registers.

- NASM does not support memory models.

  The programmer is responsible for coding CALL FAR instructions where necessary when calling external functions.

```
call (seg procedure):proc          ;call segment:offset
```

seg returns the segment base of procedure *proc*.

NASM Syntax

- NASM does not support memory models.

  The programmer has to keep track of which functions are supposed to be called with a *far call* and which with a *near call*, and is responsible for putting the correct form of RET instruction (RETN or RETF).

- NASM uses the names *st0*, *st1*, etc. to refer to floating point registers.

- NASM's declaration syntax for un-initialized storage is different.

```
stack DB 64 DUP (?)      ;ERROR
stack resb 64            ;Reserve 64 bytes.
```

- Macros and directives work differently than they do in MASM.

NASM Syntax

NASM source line:

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

The ':' is optional, which can cause problems if, for example, you misspell an instruction, e.g. **lodab** instead of **lodsb**.

Use -w+orphan-labels as a command line option to the compiler to identify these!

Valid characters in labels are letters, numbers, _, $, #, @, ~, ., and ?.
Identifier valid starting characters include letters, . , _ and ?.

Instruction prefixes supported:

- LOCK,
- REP,
- REPE/REPZ
- REPNE/REPNZ

NASM Syntax

Floating point instructions can take on two-operand forms or a single operand form:

```
fadd st1                ;This sets st0 = st0 + st1
fadd st0,st1            ;So does this.
fadd st1,st0            ;this sets st1 := st1 + st0
```

Almost any float-point instruction that references memory must use one of the prefixes **DWORD**, **QWORD** or **TWORD** to indicate what size of memory operand it refers to.

Storage directives:

DB, DW, DD, DQ and DT are used for initialized data only.

RESB, RESW, RESD, RESQ and REST are used for uninitialized.

```
        db 0x55             ;The byte 0x55
        dw 'abc'            ;0x41 0x42 0x43 0x00 (string)
        dd 0x12345678       ;0x78 0x56 0x34 0x12
zerobuf: times 64 db 0      ;Equivalent to the dup op
```

NASM Syntax

EQU defines a symbol to a constant:

```
message db 'hello, world'
msglen equ $-message
```

Address mode examples:

```
mov eax,[ebx*2+ecx+offset]
mov eax,[ebp+edi+8]
```

Constants:

Suffixes H, Q and B are used hex, octal and binary. 0x also works for hex.

```
mov eax, 0xa2                ;hex
mov eax, 777q                ;octal
mov eax, 10010011b           ;binary
mov eax, 'abcd'              ;ASCII chars 0x64636261
dd 1.2
dq 1.e+10
dt 3.14159265358979238462
```

NASM Syntax

The SEG operator returns the preferred segment base of a symbol:

```
mov ax, seg symbol   ;Load the segment base.
mov es, ax
mov ebx, symbol
```

Will load ES:EBX with a valid pointer to symbol.

(Probably wont need unless you are writing 16-bit code which has multiple segments).

To declare a far pointer to a data item in a data segment:

```
dw symbol, seg symbol
```

Local Labels begin with a '.' and are associated with previous non-local label.

```
label1                  ;some code
.loop                   ;some more code
        .jne .loop      ;jumps to previous .loop
        ret             ;Treated as label1.loop
label2
.loop
        .jne .loop      ;jumps to previous .loop
```

NASM Syntax

Single-line Macros:

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

Can be used as:

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

Which expands to:

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

Note that expansion occurs at invocation time, not at definition time, e.g.

```
%define a(x) 1+b(x)              ;b(x) used before it is
%define b(x) 2*x                 ;defined here.
```

Used as:

```
mov ax, a(8)
```

Expands to:

```
mov ax, 1+2*8
```

NASM Syntax

Overloading macros is allowed.

```
%define foo(x) 1+x              ;Single arg definition
%define foo(x,y) 1+x*y          ;Double arg definition
```

Undefining macros:

```
%undef foo
```

Multi-line Macros:

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

And use as:

```
myfunc: prologue 12
```

Expands to:

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

NASM Syntax

Conditional assembly:

Given the macro (21h is a DOS interrupt):

```
%macro writefile 2+             ;Greedy macro params
          jmp %%endstr          ;%% defines macro-local
%%str:    db %2                 ;labels which are different
%%endstr: mov dx, %%str         ;each time the macro is
          mov cx, %%endstr-%%str ;invoked.
          mov bx, %1
          mov ah, 0x40
          int 0x21
%%endmacro
```

And the call:

```
%ifdef DEBUG
          writefile 2, “I’m here”, 13, 10
%endif
```

Using the command-line option -dDEBUG, expands the macro otherwise it is left out (similar to C).

Note that “I’m here”, 13, 10 is substituted in for %2 in the above code.

NASM Syntax

Structure definitions:

```
            struc mytype          ;Defines mytype as 0
mt_long:    resd 1                ;Defines mt_long as 0
mt_word:    resw 1                ;Defines mt_word as 4
mt_byte:    resb 1                ;Defines mt_byte as 6
mt_str:     resb 32               ;Defines mt_str as 7
            endstruc
```

mytype_size is also defined as the total size, and is 39 in this example.

To declare instances:

```
mystruc:    istruc mytype
            at mt_long, dd 123456  ;Same order as given in
            at mt_word, dw 1024    ;the definition.
            at mt_byte, db 'x'
            at mt_str, db 'hello, world', 13, 10, 0
            iend
```

To reference, you must use:

```
mov eax, [mystruc+mt_word]
```

The align (and alignb) directive can be used to align the data.

NASM Examples

Hello World:

```
section .data
  msg  db 'Hello, world!',0x0A
  len  equ $ - msg  ;length of hello string.
section .text
  global _start     ;must be declared for linker (ld)
  _start:           ;we tell linker where is entry point
    mov eax, 4      ;system call number (sys_write)
    mov ebx, 1      ;file descriptor (stdout)
    mov ecx, msg    ;message to write
    mov edx, len    ;message length
    int 0x80        ;call kernel
    mov eax, 1      ;system call number (sys_exit)
    int 0x80
```

To produce hello.o object file:

nasm -f elf hello.asm

To produce hello ELF executable:

ld -s -o hello hello.o

NASM Examples

A program to process command line args:

```
section .text
  global _start      ;must be declared for linker (ld)
  _start:            ;we tell linker where is entry point
    pop ecx          ;pop argument count
    dec ecx          ;is there anything on command line?
    jz  exit         ;no, exit

    pop ecx          ;skip our name

  mainloop:
    pop ecx          ;pop an argument - message to write
    or  ecx, ecx     ;end of arguments?
    jz  exit         ;yes, exit

    mov esi, ecx     ;now we must compute length of an
                      ;argument
    xor edx, edx     ;edx will hold strlen
    dec edx
```

NASM Examples

A program to process command line args (cont):

```
.strlen:
   inc edx
   lodsb
   or  al, al
   jnz .strlen
   mov eax, 4          ;system call number (sys_write)
   mov ebx, 1          ;file descriptor (stdout)
   int 0x80            ;call kernel (ecx and edx are already
                        ;set before)
   jmp short mainloop

exit:
   mov eax, 1          ;system call number (sys_exit)
   int 0x80
```