

NewBasic Assembler
(16 bit version)
Version 00.24.94
Freeware

Forever Young Software
(C)opyright 1984-2003
All rights reserved
05 July 2001

Please read 'Precautions' near the end of this documentation. It has to do with some features that are not yet in NBASM that you will need to know about using libraries and the .external directive, as well as the 32 bit instructions.

Table of Contents

Introduction and Overview

Legal Terms and Conditions	2
What is NBASM indented for	2
What can NBASM do	2
What won't NBASM do	2
Difference Between Other Assemblers	4
Conflicts with Other Software	5
System Requirements	5
Software Requirements	5
Latest Version	6
Bug report	6
Mailing List	6
About the Author	6

Using NBASM

Running NBASM	8
Assemble-time Macros	10
Optimizer and Precautions	11
Using the Library	12

Your Source File(s)

Syntax	14
Operands	16
Memory References	21
Indirect Branching	21
Pseudo-Operations	24
Directives	29
Instructions	36
FPU Instructions	42
Structures and Struct	51
Sizeof	53

Miscellaneous Items

Errors	43
Precautions	47

Examples	48
Appendix A (undocumented instructions)	49
Appendix B (boot sector writing)	50

Introduction and Overview

Legal Terms and Conditions

This package is provided to you under the following conditions:

1. This package is distributed as freeware. You may copy the NBASM.ZIP files, and give them to anyone who accepts these terms, as long as the copies you distribute are complete and unmodified.
2. There is no registration fee for using this package. However, a note or comment saying that you have downloaded this package and are using, deleting, liking, loving, hating, etc., would be appreciated. You can send all comments to:

fys@cybertrails.com

Commercial, government, and educational institutions and training facilities must be registered in order to use NBASM. Please contact me for special terms and conditions.

3. This package may not be sold to anyone. If this package is distributed on a disk, any fees collected must be for media and handling and may not exceed ten US dollars.
4. I am not to be held liable for any damages, of any kind, arising from any failure of any programs in this package or any programs created with this package. Use at your own risk.
5. You can sell or distribute any program that you have written or modified using this assembler. There is no fee for any executable file produced with this package as long as this executable file is for non-commercial use. If you use this package for commercial use, please contact me for special terms and conditions.
6. Sorry for all the legalities. This is to protect me as well as protect you.

What is NBASM indented for?

1. NBASM is indented for users that want to learn assembly language as well as experienced users who want to create assembly language projects.
2. NBASM can be used to create small or large projects. There are no restrictions.
3. I created NBASM mostly for the enjoyment and to learn how to create an assembler. You may use NBASM to its full capabilities, but please take note that it does contain bugs and/or errors. If you find a bug and/or error, please let me know. I am constantly adding new things to NBASM and fixing bugs and errors is part of the fun.

What can NBASM do?

1. NBASM takes a text file, consisting of mnemonics, user-defined symbols, numbers, and pseudo-ops, and produces a file of corresponding machine language for the 8086/88 through the Pentium processors.
2. Your source file size is not restricted and can be any size. However, any include files that you may use must be 64k or less in size (each).
3. NBASM supports long filenames, and uses long filenames by default if the current operating system has LFN support. Use '/l' on the command line to force long filename support. If this switch is used, NBASM does not check for OS LFN support. Use '/n' on the command line to force short filename support. This switch overrides, the /l switch.
4. NBASM is capable of processing a few output file types:
 1. COM files that are ready to execute by DOS
 2. SYS files, that if correctly written, can be used for boot sectors, device drivers, etc.
 3. My own format of object files (.NBO) ready for linking to other .NBO files and/or libraries to create .EXE files ready for DOS.
* However, this format is currently not working correctly. *
5. External variables and procedures will be supported once the .NBO part has been fixed.
6. NBASM is 'fairly' fast. The source to NBASM is assembly and builds a complete COM file from a multi-source project at 100,000 lines a minute on a P133. If your source code is a single file and that file is less than 64k in size, this time is cut considerably since there is little file access.
7. NBASM is easy to use. Even though NBASM requires a few "red tape" directives, once you have these at the top of you source, there are not many more "red tape" items needed.
8. NBASM has a full function listing capability, listing output to a .lst file for your debugging purposes.
9. NBASM has many error messages to help you catch the errors in your source.

What won't NBASM do?

1. Macros are not supported. Currently, I do not have plans to support user defined macros. However, if the interest of the user warrants them, I may add this feature.
2. NBASM doesn't support any Windoze/Linux/UNIX/etc. file formats. DOS is assumed on all projects as input and output.

Difference Between Other Assemblers

Here I will try to explain the differences between NBASM and some other assemblers and how to modify your source code to work with NBASM.

While maybe some of the following assemblers support them, NBASM does not support the following:

1. User defined Macros
2. All of the x87 instructions (I hope to have the rest included soon)
3. EXE file formats (.obj's)
(NBASM will sometime soon support .nbo's which is similar to .obj's)

MASM 5.1x (Microsoft)

There are little differences between NBASM and MASM 5.1x. I have tried to make NBASM as close to MASM as I could and still satisfy my wants. If you use the newer/easier segment directives of MASM, you will have to change your source very little to assemble with NBASM.

NBASM does not support the .DATA?, .FARDATA, & .FARDATA? segment directives.

MASM	NBASM
end label	.end label
repx movsx	repx movsx
count dup(value)	dup count,val
mov byte ptr [xxxx],xx	mov byte [xxxx],xx
mov word ptr [xxxx],xx	mov word [xxxx],xx
mov dword ptr [xxxx],xx	mov dword [xxxx],xx

(along with a few other minor items)

TASM (Borland (Inprise))

I have not used this assembler. However, I have heard that this assembler has a directive to make it 'act' like MASM.

NASM (NetWide Assembler)

After looking at NASM, I have no plans to try to compare it with NBASM. I was not impressed with its syntax, even though it has a wide range of output. (This is only my opinion. Please no flames.)

CHASM 4.1x (David Whitman)

NBASM is similar to CHASM 4.1x. However, NBASM uses segment directives, the .model directive, and some other advanced features. There are a few modifications to CHASM besides the segment directives to make your source assemble with NBASM.

CHASM	NBASM
mov bx,offset(symbol)	mov bx,offset symbol
symbol ds 10,0	symbol dup 10,0
SHx reg/ROx reg	SHx reg,1/ROx reg,1
label	label:
(along with a few other minor items)	

NBASM also wants the .end directive at the end of your source.

Conflicts with Other Software

1. MCAFEE in a Win9x DOS session

MCAFEE's VSCAN for Windoze 9x makes NBASM pause just after the Copyright information that is printed to the screen. It is probably scanning the .COM file for viruses before it writes it to disk. To stop the 'pause', right click the 'V' (VirusScan) icon in the task bar and click on EXIT. Once you finish your DOS session, you can reload VirusScan. If you don't get the 'pause', then there is no need to unload VirusScan.

NBASM does not contain any intentional virii. However, it is always safe to scan all files you get from the internet or any type of disk.

No other known conflicts. Please let me know if you find one.

System Requirements

1. a 186 or higher Intel based system (x86)
2. DOS 2.0 or higher.
3. a CGA or better monitor (Monochrome should be fine too. :))
4. a disk drive with a minimum of 128k of free disk space (is that all?) plus room for the .COM and .LST file(s).
5. a minimum of 145k free memory plus the following:
 - the size of the source file (up to 64k more)
 - the size of the included file(s) used (up to 64k more each)
6. a keyboard (Duuhh! =:o)
7. a printer capable of text printing (for hard-copies right?)

Software Requirements

1. as stated above, DOS 2.0 or higher (NBASM works in a Windoze (95) DOS session just fine)
2. some sort of viewer to view this document. (But you are reading this right now, so you got at least that much, right?)
3. a similar editor to create DOS ascii text files. (EDIT.COM|EXE)
4. PKUNZIP/WINZIP to unzip the zip file. Again, you are reading this file, so you must have one of these :0)

Latest Version

You can get the latest version of NBASM and utilities at:

<http://www.cybertrails.com/~fys/newbasic.htm>

Bug Report

NBASM is not perfect and there are bound to be bugs in it.

Please report bugs to:

fys@cybertrails.com

<http://www.cybertrails.com/~fys/newbasic.htm>

Known Bugs/Errors

Please see the included README.TXT file for a list of known issues.

Mailing List

I have an announcement mailing list which I use to announce news and updates about NBASM and other items pertaining to NBASM. It is a moderated list and will only be used for NBASM announcements.

If you would like to receive mail each time I make an update or announce something about NBASM, please send an empty email message to:

nbasm-subscribe@yahoogroups.com

To unsubscribe from the list, please send an empty email message to:

nbasm-unsubscribe@yahoogroups.com

You can also view the archived messages at:

<http://groups.yahoo.com/group/nbasm>

If you have any problems with the above mailing list, please let me know at:

fys@cybertrails.com (Please include your Yahoogroups login in name)

I don't plan on more than about a single message per week at the most. So don't worry about me filling your "inbox" :-)

About the Author

I am a full time Freelance DOS and Windows 9x programmer. I have been programming since 1984 and most of these projects for the Intel x86 systems.

I have been making a nice living from my work, and with your much appreciated support, I will continue to improve my products, and enhance them with new features and capabilities.

I can be reached at the following addresses:

<http://www.cybertrails.com/~fys/index.htm>
fys@cybertrails.com

Forever Young Software
%Benjamin David Lunt
PO Box 875
Taylor, AZ 85939-0875

Using NBASM

Running NBASM

From the DOS prompt, type: (Items in brackets are optional.)

```
NBASM source[.asm target.(COM|NBO|SYS|BIN) parameter list]
```

source is the filename of your source file. If you do not include an extension, .asm is assumed.

target if given is the name that NBASM will name your target file.

If not given, NBASM will use "source" as the target filename with The .COM extention.

parameter list can be any one of the following in any order.

Please note that any parameter are first come, first serve and will overwrite any parameters previously listed on the command line.

(the '-' char can be substituted for the '/' char)

** All parameters are case sensitive **

- /1 - do pass 1 only
does not create a target file. For syntax checking.
- /a - align segments in alphabetical order
- /b - beep if errors were encountered during assembling
- /d - do not print diagnostic errors
- /h - help screen
- /i{} - include path (if {} only, then start from root dir)
specify an include path between the {}'s
- /j - same as the .jumps directive
- /l - Force Long Filenames
force NBASM to use long filenames. If this switch is used NBASM does not check for long filename compatibility.
(NBASM uses long filenames by default)
- /n - Force Short Filenames
force NBASM to use short filenames. If this switch is used NBASM does not allow long filenames.
- /o - optimize for size
- /s - check syntax only
does not create a target file. For syntax checking.
- /w<d> - Tells NBASM to write your code to drive <d> as a boot sector
(see the section on boot sectors in Appendix B)
- /x - Tells NBASM to create a list file
(the list file as the .LST extension)

NBASM makes two passes over your source file, outputting the listing and object code on the second pass.

NBASM returns an error code in ERRORLEVEL for use in .bat files. The following codes are returned:

- 00h - No error encountered
- 02h - Error opening file
- 03h - Invalid path used with /I{} parameter
- 06h - Target File can not be Source File

08h	- Not enough memory
0Bh	- Invalid command line
0Eh	- Need DOS 2.0+ or an 80186 or higher CPU
0Fh	- Trying to include an already nested include file
13h-FEh	- error code of last error found in source code
FFh	- Undefined error

Assemble-time Macros

These macros are assembler time macros. All macros start with the percent sign (%). Each macro can have one or more parameters. Some macros may require an ending macro declared as the same macro name but with two percent signes (%%)

1. %DATA

This macro puts the segment of DS into the operation as a word value for use in .EXE files. Used only in the small model.

Example:

```
mov ax,%DATA
mov ds,ax
```

2. %ERROR "pass_number" 'text to print'

This macro outputs text to the stdout device (screen) when the assembler gets to this location in the source. Prints what is on this line only, and requires no ending macro name. The "pass_number" can be 1 or 2. If anything else, doesn't print.

'text to print' must be any assembler compatible string.

This macro also increments the ERROR count.

Example:

```
%ERROR 1 'Print this line on pass one only'
```

3. %LINE

This macro returns an immed16 (word) denoting the current line number that this macro is on.

Example:

```
mov ax,%line ; this line is on line: %line
```

4. %OUT "pass_number" 'text to print'

This macro outputs text to the stdout device (screen) when the assembler gets to this location in the source. Prints what is on this line only, and requires no ending macro name.

The "pass_number" can be 1 or 2. If anything else, doesn't print.

'text to print' must be any assembler compatible string.

5. %PRINT symbol

This macro prints a 32-bit hex value of the symbol given. The symbol can be just about any symbol, math function, immediate, etc.

6. %TITLE 'title text'

This macro outputs the title to the .NBO file if model is small.

If the model is tiny, this macro is ignored.

7. %VER

This macro returns an immed16 (word) denoting the current version of the assembler. High byte is major version and low byte is minor version.

Example:

```
mov ax,%ver ; get the current version of the assembler
```

Optimizer and Precautions

NBASM includes an optimizer with different levels of optimizations. Currently, there is only one level implemented.

Level 0: optimize for size

If you include the /o switch on the command line and/or use the .OPTON/.OPTOFF directive, then NBASM will optimize in the following areas:

1. Optimization #1:

```
if      MOV  REG16,00
then    XOR  REG16,REG16
```

example:

```
if      MOV  DX,00h
then    XOR  DX,DX
```

notes/precautions:

If you are relying on the flags register from a previous instruction and the optimizer changes the above item, the flags register will be destroyed by the XOR instruction. If you want to use MOV dest,00h so that the flags won't be affected by the instruction then do the following example:

```
AND  AX,45h      ; want to use flags later
.OPTOFF
MOV  DX,00h      ; MOV doesn't change the flags
.OPTON
JZ   ....       ; use flags from AND above
```

2. Optimization #2:

```
if      CMP  REG,00
then    OR   REG,REG
```

example:

```
if      CMP  DX,00h
then    OR   DX,DX
```

notes/precautions:

Even though NBASM will change an 8 bit register above, there is no size difference gained. It is done for consistency.

3/4. Optimization #3/#4:

```
if      ADD  REG/MEM,1   or  SUB  REG/MEM,1
then    INC  REG/MEM     DEC  REG/MEM
```

example:

```
if      ADD  AX,01      or  SUB  AX,01
then    INC  AX          DEC  AX
```

notes/precautions:

Please note that SUB and DEC do not effect the flags in the exact same way.

5. Optimization #5:

```
if      INT  20h
then    RET
```

notes/precautions:

only works in .tiny model (.COM files) with out the use of the .start directive. Also, make sure that you know that [SP] = 00h. Use .OPTOFF if you are not sure.

Using the Library

NBASM includes a library for Tiny Model.

For the Tiny model, the library NBASMC.LIB is included. It is a library of binary code that NBASM appends parts of it to the end of the .com file when the .external directive is used.

For example, the following source would be assembled and then NBASM would include the binary code for 'prtstring' at the end of the .com file from the NBASMC.LIB library file:

```
.model tiny          ; create COM file
.186                 ; allow 186 instructions
.external prtstring  ; include the code for prtstring
.code                ; start of code segment
    push offset msg1 ; use library function (prtstring)
    call prtstring   ;
    int  20h         ; exit to DOS

msg1    db  'This string gets printed to the screen',0

.end
```

Note: If a symbol is declared external via .external but is not used in the code, NBASM will NOT add the associated code to the end of the .com file. For this it is safe to declare all external procedures you might use and not have to worry about them taking up unnecessary space in the .COM file. However, they will take up space in the symbol table at assemble time.

Note: When LIBMAN creates a library, it adds a CRC to each function added. NBASM checks this CRC to make sure that the library function wanted is not corrupted before it adds the necessary binary code.

See <http://www.cybertrails.com/~fys/newbasic.htm> for more about LIBMAN.

Please note that all code that is used/called from a library with this technique is added to the end of your .com file. So if you plan to use the remaining memory after your .com file, please take in mind the size of the code included by the library, or you might overwrite the code and crash the machine.

I plan to add some code and documentation to NBASM to allow the user to find where the end of this added code will be. Until then, be careful about assuming where the code is and how long it is.

Please see Example #2 at the end of this documentation for another example on the .external directive. Also see DEMO1.ASM for another example.

Please read the documentation for the library on how to use each procedure included in the library(s). This documentation is in NBASMC.DOC included with this package.

For the Small model, the same library NBASMC.LIB is used. However, this part is not finished yet and is not documented until then.

Your Source File

Syntax

NBASM accepts a standard DOS text file for input. Lines may be any combination of upper and lower case characters. NBASM does not distinguish between the two cases. Every thing except quoted strings are automatically converted to upper case during the parsing process. Thus, SYMBOL, Symbol, and symBol all refer to the same symbol.

The following characters are reserved by NBASM:

space	(ascii 32)
comma	(ascii 44)
single quote	(ascii 39)
double quote	(ascii 34)
semi-colon	(ascii 59)
underscore	(ascii 95)
dollar sign	(ascii 36)
percent sign	(ascii 37)

Note: Most editors, including Microsoft's EDIT, won't allow the following ascii characters: ascii 0, 9, 10, and 13

For this reason, NBASM will change these four (4) chars to ascii 32 (space) if found in your source code. This happens in all lines, INCLUDING quoted strings. So if you want to use these four (4) chars in a quoted string, you must use the ascii value as a literal.

Example:

```
string db 00,'',09,10,''
```

',13,'

□ etc...

Each source line must be less than 256 characters long and have the following format:

```
Label: Operation Operand(s) ; comment
```

The delimiter, which is a space or a comma, separates the different fields of an input line. Any number of either delimiter may be used to separate fields.

Explanation of Fields

Label: A label is a string of characters, beginning in column 1. Depending on the operation field, the label might represent a program location for branching, or a memory location. Labels can be as many characters long as wanted, though on average, NBASM allows about 1500 thirteen byte symbols.

NOTE: Anything beginning in column 1, except a comment or directive, is considered a label.

NBASM allows the use of the 'Unique' Label directive called, '@LABEL'. When NBASM comes to this directive, it will do one of two things depending if this will be a label or symbol. If it is to be a label, then NBASM will put a 'Unique' name as the label. NBASM starts with 'NB0000' as the label and then increments to 'NB0001' for the next and so on.

If you put a label that has this name and also use @LABEL, you might receive errors.

If @LABEL represents a symbol other than a label, NBASM will put the current 'unique' name in as this symbol name. So with this technique, every '@LABEL' reference that is not in column 1 will point to the next label that is in column 1 AND named '@LABEL'. This allows you to use more than one to point to the same label reference. Please note that this will only work in the forward direction. See Example #1 at end of this document for more information.

Operation: Either a pseudo-op or an instruction mnemonic.

Operand(s): A list of one or more operands, as defined on page 16, separated by delimiters.

Comment: Any string of characters, beginning with a semicolon. Anything to the right of a semicolon will be ignored.

If you want to have a lot of text and don't want to start each line with a semicolon, then use the COMMENT <char> ... <char> command. The word COMMENT must start in the first column of the line and then the following char (after the space) will denote the start of the comment. NBASM will start with the NEXT line and search for that char again. Once found, assembly starts on the NEXT line after the found char.

Please Note: COMMENT <char> ... <char> will not work on the same line. Use the semicolon (;) for this.

Example:

```
COMMENT # all the text and code that is on this line
        and this line
        and this line
# and any thing on this line will NOT be assembled.
; ASSEMBLY resumes here
```

Operands

The following operand types are allowed

Immediate data: A number, stored as part of the program's object code. Immediate data is classified as either byte; expressible as an 8 bit binary integer; word, expressible as a 16 bit binary integer, or dword, expressible as a 32 bit binary integer. If context requires it, NBASM will left-pad byte values with zeros to convert them to word values and the same with word to dword values.

Attempts to use a word value where only a byte value will fit will cause an error message to be printed, and the same with dword values.

Immediate data may be represented in 7 ways:

1. A signed decimal number in the range of
-2147483648 to 2147483647 (signed)
0 to 4294967296 (unsigned)

Example:

```
MOV AL,22
MOV BL,-15
MOV BX,-1000
MOV EBX,-1000
MOV DX,65535
MOV EDX,-1
MOV EDX,12345678
```

2. A series of up to 8 hex digits, followed by the letter H (lower or upper case). All hex numbers should start with a numbered digit of 0-9 and should be in the range of: 0 to FFFFFFFF

Example:

```
MOV CL,0FFFFFFFH ; -1
MOV CL,-1H ; -1
ADD DL,0FDh
MOV CX,1234H
MOV CX,-1234H
MOV ECX,12345678H
```

3. A series of up to 32 binary digits (0|1), followed by the letter B (lower or upper case). Since 32 bits is hard to keep track of, you may use a spacer (_) to help you keep track of each part of a bit map.

Example:

```
AND CL,00010010b
MOV CX,1100000011111111b
MOV CX,11101010_11010111b ; <- spacer used
MOV CX,11_0000_111_0000_111b ; <- spacer used
MOV ECX,11101010_11010111_01100101_01010111b ; <- spacer used
MOV ECX,11000000111111110100000011111111b
```


4. A symbol representing the types above defined using the EQU pseudo-op.

Example:

```
TEMP EQU $           ; temp = this location
MASK EQU 10h
MASK1 equ 1000h

    or    CL,MASK
    xor   ax,mask1
```

5. The offset of a label or storage location returned by the OFFSET operator. OFFSET always returns a word value. OFFSET is used to get the address of a named memory location, rather than its contents.

Example:

```
mov di,offset Buffer
```

6. The ASCII value of a printable character, represented by the character enclosed in single or double quotes. Thus, the following lines will generate the same object code:

```
MOV AL,65 ;ascii code for 'A'
MOV AL,'A'
MOV AL,"A"

; these two lines are the same
MOV AX,4142h
MOV AX,'AB'

; these two lines are the same
MOV EAX,41424344h
MOV EAX,'ABCD'
```

7. In the form of a math equation enclosed in ()'s. You can use any form of a constant number and or EQUate and use the following operators:

Note that all numbers are considered 32-bit. This means that even though 0FFFFh would represent -1 if words were used, 0FFFFh = 65535 in any math function. If you want -1, use one of the following:
-1, 0FFFFFFFh, 11111111111111111111111111111111b, or ~0

The following items allow 32bit numbers as both the operand and the result:

+	add	(32bit add 32bit)
-	subtract	(32bit sub 32bit)
	or	(32bit or 32bit)
^	xor	(32bit xor 32bit)
&	and	(32bit and 32bit)

The following items allow 32bit numbers as only the number to be worked on. The "work with" number must be 16 bit and be a positive integer:

- * multiply (32bit mul 16bit)
- / divide (32bit div 16bit)
- > shr (32bit shr 16bit)
- < shl (32bit shl 16bit)

The following item allows only one operand as a 32bit number:
~ not (not 32bit)

Negative integers are allowed in the first 5 operators above. Spaces are allowed between operands.

The "offset" directive is not allowed within a math symbol. Simply give the name of the symbol without the "offset" directive for the same effect.

Example:

```

MOV AX, (1+2*3/4|5^6&7>8<9) ; 0
MOV AX, (-1+2) ; 1
MOV AX, (-1 + 2) ; 1
MOV AL, (~0 + 1) ; 0 (~0 = -1) (-1 + 1 = 0)
MOV AX, (~1 + 1) ; FFFFh (~1 = FFFEh) (FFFEh + 1 = FFFFh)
MOV AX, (1--1) ; 2
MOV AX, ('0' - 48) ; 0
MOV AX, ( 100h > 1 ) ; 80h
MOV EAX, ( 100h > 1 ) ; 80h
MOV EAX, (123456h+123456h) ; 2468ACh
MOV AX, (TEMP+100h) ; TEMP+100h (where TEMP is an EQUate)
MOV AX, (TEMP+100h) ; offset TEMP+100h (where TEMP not an EQUate)
MOV AX, (TEMP-TEMP1) ; offset TEMP - offset TEMP1

JMP SHORT ($+2) ; jump to next instruction

```

Please Note: Currently, you can not nest this form of operand:
i.e: mov ax, (1+(1+(1+1))) is NOT allowed

A note about the NOT operator (~) above: When you use the NOT operator, the whole 32bit value is NOT'ed. Look at the following:

```

mov al, (0FFh+1) ; returns an error ( 100h won't fit in a byte )
mov al, (~0+1) ; (no error) returns 00h (~0 = -1. -1 + 1 = 0)

```

Register Operands:

- An 8 bit register:
AH, AL, BH, BL, CH, CL, DH, DL
- A 16 bit register:
AX, BX, CX, DX, SP, BP, SI, DI
- A 32 bit register:

EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI

A segment register:

CS, SS, DS, ES, FS, GS

(FS and GS are .386+)

Memory Operands:

The contents of a memory location addressed by one of the following methods. Note that none of the memory addressing options specifies that whether a byte, word, or a dword operand is being referenced when used with a register since a register already has a size.

Direct address

A number, or symbol representing a number, enclosed in brackets, indicating an offset into the data segment.

Example:

```
Buffer EQU 2222h
MOV BH,[Buffer]
MOV [80h],DI
```

A symbol defined to be a variable (i.e. a named memory location) using the EQU pseudo-op.

Example:

```
CmmdLen EQU [81h]
MOV DI,CmmdLen
```

A symbol defined to be a variable by its use with a storage defining pseudo-op.

Example:

```
MOV AX,FLAG
MOV Pos,BX
FLAG dw 22
Pos dw 22h
```

Indirect Address:

The address of the operand is the sum of the contents of the indicated register(s) and a displacement. The register, or sum of registers, are enclosed in square brackets: []

The displacement is optional, and takes the form of an immediate operand, placed without intervening delimiters to the left of the first bracket or at the end before the last bracket.

Displacements in the range -128 to 127 (hex 00 - 7F, FF80 - FFFF, FFFFFFFF80-FFFFFFFF) are interpreted as signed 8 bit quantities. All other displacements are interpreted as unsigned 16 bit quantities.

Note that although the 8086/88 supports unsigned 16 bit displacements up to hex FFFFh for indirect addressing, NBASM does not distinguish between -1 and 0FFFFh. They are both the same value to NBASM.

The following indirect modes are allowed:

Indirect through a base register (BX|BP), index register (SI|DI), memory operand, immediate, or a combination of operands.

*** Please Note: Currently, the 32 bit indirect addressing using EBP, ESP, [base32+index32], and any scaling (*1,*2,*4,*8) may produce unpredictable results.

Examples:

```
mov  al,[bx+1]
mov  al,[bx][1]
mov  al,[si+1]
mov  al,[si][1]

mov  ax,[bx][si][-10h]
mov  ax,[bx][si-10h]
mov  ax,[bx][10h][di]
mov  ax,[10h][si][bx]
mov  ax,10h[di][bx]
mov  ax,+10h[di][bx]
mov  ax,-10h[di][bx]
mov  ax,-10h+[di][bx]
mov  ax,[bp][10h][si]
mov  ax,[bp][di][10h]
mov  ax,[si][bp][10h]
mov  ax,[di][bp][10h][10h]

mov  ax,[bx+100h+05h]
mov  ax,[bx+0]
mov  ax,[bp+1]
mov  ax,[bx+1-1]
mov  ax,[bx+si]
mov  ax,[bx+si+1]
mov  ax,[bx+di]
mov  ax,[bx+di+1]
mov  ax,[bp+si]
mov  ax,[bp+si+1]
mov  ax,[bp+di]
mov  ax,[bp+di+1]

mov  ax,[05h]
mov  ax,[100h]
mov  ax,[100h+05h]
mov  ax,[105h]

mov  ax,[memory]
mov  ax,[memory+1]
mov  ax,[memory+bx]
mov  ax,[bx+memory]
mov  ax,[bx+memory+1]

mov  ax,memory
mov  ax,memory[bx]
mov  ax,memory[bx+1]
```

```
mov ax,memory[bx+0]
mov ax,memory[bx+1]
mov ax,memory[bx+si]
mov ax,memory[bx+si+0]
mov ax,memory[bx+si+1]
mov ax,memory[bx+si+1+1]
```

Labels

A label on a machine instruction may be used as an operand for call and jump instructions.

Example:

```
DoIt PROC NEAR
CALL DoIt
JMP Place
```

Strings

A string is any sequence of characters (including delimiters) surrounded by single or double quotes. Note: Empty strings are ignored

Example:

```
DB 'Forever Young Software'  
DB "Copyright 1984-2001"  
DB '' ; this is ignored  
DB 12,',' ,30 ; the empty string is ignored  
; returns 12,30
```

Memory references

When one specifies the address of a memory location, it is unclear how large an operand is being referenced. An operand might be a byte, word, or a dword. If a register is present as an operand, it is assumed that the memory operand matches the register in size. Exceptions to this rule are the shift and rotate instructions, where the CL register is used as a counter, and has nothing to do with the size of the other operand.

Example:

```
MOV MASK,AX ;mask is a word  
MOV AL,[BX] ;BX points to a byte  
AND [SI] ;error, operand of unknown size  
SHR MASK,CL ;error, mask is of unknown size
```

If no register is present, (or if the only register is CL being used as a counter) the size of the memory operand is specified by adding the suffix "B", "W", or "D"; or a type specifier "BYTE", "WORD", or "DWORD" to the instruction mnemonic.

Examples:

```
INCB [BX] ;BX points to a byte  
INC BYTE [BX] ;BX points to a byte  
INCW [BX] ;BX points to a word  
INC WORD [BX] ;BX points to a word  
INCD [BX] ;BX points to a dword  
INC DWORD [BX] ;BX points to a dword  
INC DWORD [EBX] ;EBX points to a dword  
MOVW TEMP,00H ;TEMP is a word  
MOV WORD TEMP,00H  
MOVD TEMP,00H ;TEMP is a dword  
MOV DWORD TEMP,00H ;TEMP is a dword  
MOVB DH,[BX] ;error, register already specifies size
```

Indirect Branching

The 8086/88 supports two flavors of indirect branching. Intra, and inter segment. A register is set to point at a memory location which contains a new value for the program counter, and in the case of intersegment branching, a new value for the CS register as well.

Please note: If a type of NEAR or FAR is not specified, NEAR is assumed.

Examples:

```
CALL [BX]          ; intrasegment memory indirect call
CALL [EBX]         ; intrasegment memory indirect call
CALLN [BX]         ; intrasegment memory indirect call
CALL NEAR [BX]     ; intrasegment memory indirect call
CALLF [BX+SI]     ; intersegment memory indirect call
CALL FAR [BX]     ; intersegment memory indirect call
CALL FAR [EBX]    ; intersegment memory indirect call

CALL AX           ; intrasegment register indirect call
CALLN DX          ; intrasegment register indirect call

JMP [BP+DI]       ; intrasegment memory indirect jump
JMPN [BP+DI]      ; intrasegment memory indirect jump
JMP NEAR [BP+DI] ; intrasegment memory indirect jump
JMPF [DI+4]       ; intersegment memory indirect jump
JMP FAR [DI]      ; intersegment memory indirect jump

JMP CX            ; intrasegment register indirect jump
JMPN CX           ; intrasegment register indirect jump
JMPS CX           ; intrasegment register indirect jump

JMP offset,segment ; jump 'very far' :-)
JMP FAR offset,segment ; jump 'very far'
```

Please note:

Even though the next two lines assemble, and DEBUG correctly disassembles them, they would not create correct branching.

```
CALLF DX ; intersegment register indirect call
JMPF CX ; intersegment register indirect jump
```

Long and Short Jumps

Two types of relative jumps are supported by the 8086/88: Short (specified by a signed 8 bit displacement) and Long (specified by a 16 bit displacement). Both are implemented in NBASM as a jump to a label.

The short jump is specified by mnemonic JMPS or JMP SHORT. Since one of the displacement bits is used as a sign bit, only seven bits are left to express the magnitude of jump. JMPS (and similarly, all the jump on condition instructions) is thus limited to branching to labels within a range of -128 to +127 bytes.

Example: (non conditional)

```
Start:
  JMPS Start ;short jump (2 bytes) (intrasegment)
  JMP SHORT Start ;short jump (2 bytes) (intrasegment)
  JMPS End ;short jump (2 bytes) (intrasegment)
```

```
JMP SHORT ($+2) ;short jump (2 bytes) (intra segment)
JMP End        ;long jump (3 bytes) (intra segment)
JMP Start      ;long jump (3 bytes) (intra segment)

JMP offset,segment ; jump 'very far' (intersegment)
JMPF offset,segment ; jump 'very far' (intersegment)
JMP FAR offset,segment ; jump 'very far' (intersegment)
```

If you are using the .386 or higher directive, NBASM will allow a conditional near jump. A condition jump that is more than 128 byte displacement. However, this now uses the four (4) byte instruction instead of the two (2) byte instruction. If you know that the jump will be less than 128 bytes in displacement, then include the SHORT directive on the line.

Example: (conditional)

```
Start:
.286 ; (or lower)
JC Start ;short jump (2 bytes)
JC SHORT Start ;short jump (2 bytes)
.386 ; (or higher)
JC Start ;short jump (4 bytes)
JC SHORT Start ;short jump (2 bytes)
```

Instruction Prefixes.

The 8086/88 supports three instruction type prefixes:

Segment override

An alternate segment register is specified for a reference to memory. (CS:, DS:, ES:, SS:, FS:, or GS:)

Repeat

REP, REPE, REPNE, REPZ, REPZ

A string primitive is repeated until a condition is met.

LOCK

Turns on the LOCK signal. Only useful in multiprocessor situations (8086/88 and 8087 for example).

NBASM implements these prefixes as separate instructions, rather than prefixes to another instruction. Some must appear on the same line while others on a separate line, immediately before the operand they modify.

Example:

```
ES: ;
MOV AX,FLAG ; flag is in the extra segment
```

```
MOV AX,ES:FLAG ; this line is the same as the two above
REP           ; must appear on a separate line just above
MOVSB        ; move bytes until CX decrements to 0
```

Pseudo-Operations

The following pseudo-ops are implemented:

\$: This location

The '\$' symbol can be used as an operand to find the offset of a position relative to the start of the segment.

Example: (code segment)

```
mov ax,$
mov ThisLoc,ax ; ThisLoc = offset of POS in segment
ThisLoc equ $
```

Example: (data segment)

```
ThisPos dw $
ThisPos db $ ; returns an error. $ is sizeof word
```

?: Undefined Data

The '?' denotes undefined data in a DUP line. (see below)
However, if you use the '?' symbol else where, NBASM places an undefined value in its place as the size noted.

Example:

```
mov ax,? is the same as mov ax,xxxxh
mov al,? is the same as mov al,xxh
TEMP DB ? is the same as TEMP DB xxh
TEMP DW ? is the same as TEMP DW xxxxxh
```

DB: Declare Byte(s)

Memory locations are filled with values from the operand list. Any number of operands may appear (up to the line length limit), but all must fit on one line. Acceptable operands are numbers between 00h and 0FFh (0-255 decimal), or strings enclosed in single or double quotes. You can use any ascii char from 00d to 255d except 0, 9, 10, and 13. If you need to use one of these chars, you must give the ascii value instead of the literal char.

If a label appears, it is considered a variable, and the location may be referred to using the label, rather than an address. The reason for the use of single or double quote is so that you can put a single quote as a char in a string delimited by a double quote and vice-versa.

Example:

```
MASK DB 00h,01h,02h,03h
Strg1 DB 'A string'
Strg2 DB "A string"
```

```
DB "A string's single quote"  
DB 'Can NOT use ascii char 13d. Ascii 13d =',13  
Strg5 DB ?,? ; declares two undefined bytes
```

DD: Declare double word(s)

Identical to DW (below) except uses dwords instead of words, and does not allow strings as DB and DW does. Use a range of 0 and 0FFFFFFFh

Example:

```
Val5 DD ? ; declares an undefined dword  
DD offset Strg5 ; 0000xxxxh  
DD 12345678h  
DD 'Not allowed and will return an error'
```

DW: Declare word(s)

Identical to DB (above) except uses words instead of bytes. Use a range of 0 and 0FFFFh (0-65535 decimal)

Example:

```
Val5 DW ?,? ; declares two undefined words  
DW offset Strg5  
DW 1234h  
DW 'Allowed and will NOT return an error'
```

DUP: Declare Storage (Initialize data)

Used to declare large blocks of identically initialized storage. The first operand is required, a number specifying how many bytes are declared. If a second operand in the form of a number 00h-FFh appears, the locations will all be initialized to this value. If the second operand is not present, locations are initialized to 00h. As with DB, any label is considered a variable. To save space, the object code does not appear on the listing.

Example:

```
DUP 22 ; 22 locations initialized to 00h  
DUP 100h,22h ; 256 locations initialized to 22h  
TEMP DUP 22 ; 22 locations initialized to 00 and  
; pointed to by TEMP  
DUP 0 ; error: 55h
```

DUP: Declare Storage (Uninitialize data)

This is the same as the 'initialize data' above except that it doesn't write anything to the executable file. You can specify uninitialized data at the end of your .com files this way. However, if you specify any other data or code that is not uninitialized data after an 'uninitialize data' line, NBASM will give an error of 5Ch.

Example:

```
TEMP DUP 10,? ; Uninitialize data. This only increments the 'IP'  
; position; Doesn't write to the COM/NBO file.
```

If you use any .external procedures in your .COM file, any undeclared data will be overwritten with the external code

I plan to add some code and documentation to NBASM to allow the coder to find where the end of any external code will be. Until then, be careful about assuming where the code is and how long it is.

ENDP: End of Procedure
See PROC (below) for details.

EQU: Equate

Used to equate a symbolic name with a number. The symbol may then be used anywhere the number would be used. Use of symbols makes programs more understandable, and simplifies modification.

An alternate form of EQU encloses the number in square brackets: []. The symbol is then interpreted as a variable, and may be used as an address for memory access. This version is provided to allow symbolic reference to locations outside the program segment.

Warning: Difficult to debug errors may result from using a symbol prior to its being defined by EQU. All Equates should be defined at the beginning of the code.

Example:

```
MOFFSET EQU B000H  
MONOCHROME EQU [0000H]  
LENGTH EQU (var2-var1) ; bytes between var1 and var2  
LENGTH EQU ($-TEMP1) ; Current Location - offset TEMP1
```

HIGH: Return the high byte of an expression.
Returns the high byte of the next operation:

Example:

```
mov al,high 1234h = mov al,12h  
mov al,high offset var = mov al,(high byte of offset)  
mov ax,high 1234h = mov ax,12h  
mov eax,high 1234h = mov eax,12h
```

Remember that you can not use this operator on variables.

For Example:

```
mov al,high var
```

will return the high byte of the OFFSET of var, NOT the value of var.
(also see LOW)

INCLUDE: Include a file

NBASM supports include files nested up to 10 max. An include file can have any name and extension. The file can contain most anything that a main module would. If you use an include statement and file, you must put the include file after any of the following directives:

.model, .stack, .186 (or similar), and .start

Syntax for the include statement is:

(The INCLUDE statement must start in column one as a label would)

INCLUDE filename.inc

NBASM searches for the include file in the following order:

1. NBASM uses the directory specified with the /I{} parameter if it is specified on the command line.
2. NBASM searches the path specified by the command line.

e.g.:

If you use the following command line:

NBASM demo

then NBASM searches the current directory for the include file.

If you use the following command line:

NBASM C:\source\demo

then NBASM searches the C:\SOURCE directory for the include file.

3. NBASM searches the directory that is pointed to by your INCLUDE= parameter in the ENVIRONMENT.

You can use the /I{} parameter at the command line to tell NBASM what path to look in for the include file. If you use this /I{} parameter, then NBASM does not look in the dir(s) pointed to by the INCLUDE=environment variable.

Example:

NBASM demo.asm /i{c:\includes}

This will tell NBASM that the include file is in the directory:

c:\includes\

An included file must be 65535 bytes or less and end with the .end directive. NBASM will still assemble the file(s) correctly, but will return an error if the .end directive is not used in the include file(s).

NBASM will not allow including an already nested include file.

LOW: Return the low byte of an expression

Returns the low byte of the next operation:

Example:

```
mov al,low 1234h      = mov al,34h
mov al,low offset var = mov al,(low byte of offset)
mov ax,low 1234h     = mov ax,34h
mov eax,low 1234h    = mov eax,34h
```

Remember that you can not use this operator on variables.

For example:

```
mov al,low var
```

will return the low byte of the OFFSET of var, NOT the value of var.
(also see HIGH)

ORG: Origin

Allows direct manipulation of the location counter during assembly. By default, NBASM assembles code to start at offset 100h, the origin expected by COMMAND.COM for .COM programs. By using ORG, you may override this default.

Example:

```
ORG 0 ; Code will be assembled for starting offset of 00h
```

PROC ...ENDP: Procedure Definition

Declares a procedure. One operand is required on PROC, either the word NEAR, or the word FAR. This pseudo-op warns NBASM whether to assemble returns as intra (near) or intersegment (far). Procedures called from within your program should be declared NEAR. All others should be FAR. ENDP terminates the procedure, and requires no operands. Procedures can not be nested.

The label is not required on the ENDP line.

The word USES may follow the NEAR|FAR word to tell NBASM to save the specified registers listed after the word USES.

Example:

```
MAIN PROC NEAR USES AX BX SI
...
...
RET
MAIN ENDP ; the label 'MAIN' is not required here
```

This example will declare a near procedure and save the following registers: AX, BX, and SI. Then NBASM will restore these registers just before the RET is encountered. (or a .NORET directive is encountered. See .NORET for more information)

If you do not place a RET with in your PROC/ENDP block, NBASM will return an error. See the .NORET directive for more information if you do not what a RET inside your PROC/ENDP block.

You can use the ALL and ALLF keywords. The ALL keyword is the same as if you used PUSHA/POPA (186+) and the ALLF keyword is as if you used PUSHA/POPA and PUSHF/POPF.

Please Note: Saves only the registers PUSHA saves and requires .186+

You can also use the ALLD and ALLDF keywords. The ALLD keyword is the same as if you used PUSHAD/POPAD (386+) and the ALLDF keyword is as if you used PUSHAD/POPAD and PUSHPD/POPFD.

Please Note: Saves only the registers PUSHAD saves and requires .386+

```
MAIN PROC NEAR USES ALL ; save all registers (Pusha)
or
MAIN PROC NEAR USES ALL ds ; save all registers and ds
```

```
    or  
MAIN  PROC  NEAR USES ALLF  ; save all registers and the flags
```

```
    or
MAIN  PROC  NEAR USES ALLD  ; save all registers (Pushad)
    or
MAIN  PROC  NEAR USES ALLDF ; save all 32 bit regs and eflags
```

Note: See the documentation for PUSHA(D)/POPA(D) for the registers ALL(D) uses.

If a RET is encountered outside of a declared procedure in a .COM file, NBASM will put the opcode value of 0C3h (ret) so that you can exit with one byte. If this happens in an .NBO (.model small) file or if the .start directive is used, and error will be given.

SIZEOF: get sizeof type
Return the size of a specified type.
See page 53 for more information.

ST: Structure Declaration
Declare a symbol of type struct predefined structure type declaration.
See page 51 for more information.

Directives

```
.8086
  Enables usage 8086 instructions (default)
  (for returning to 8086 after .186, .286, etc.)

.186
  Same as .8086 above and allows 80186 instructions

.286
  Enables all non-privileged instructions up to the 80286

.286P
  Same as .286 above and allows all 80286 privileged instructions

.386
  Enables all non-privileged instructions up to the 80386

.386P
  Same as .386 above and allows all 80386 privileged instructions

.486
  Enables all instructions up to the 80486 including all privileged
  286/386 instructions.

.586
  Enables all instructions up to the 80586 (Pentium) including all
  privileged 286/386 instructions.
```

Please note: Some of the instructions that NBASM allows with this directive are use for a Pentium Pro or Pentium II/III.

```
.87
  Enables usage 8087 instructions
```

(for returning to 8087 after .187, etc.)

.187

Same as .87 above and allows 80187 instructions.

(Please note: Will give error if not in .186+ mode)

.287

Same as .187 above and allows 80287 instructions.
(Please note: Will give error if not in .286+ mode)

.387

Same as .287 above and allows 80387 instructions.
(Please note: Will give error if not in .386+ mode)

.ALPHA

Tell NBASM to write the segments in alphabetical order. Used only in NBO files. If in a COM file, it is ignored.

.CODE

Specifies that all that follows will be put in the code segment until a .DATA or .END is reached. The .CODE directive must be in both types of output files and before any actual code statements.

.DATA

Specifies that all that follows will be put in the data segment until a .CODE or .END is reached. The .DATA directive can not be in a COM file (.model tiny (see below)).

.DOSEG

Specifies to the assembler to include all segments in this order:
CODE (DGROUP), DATA, BSS, STACK (default)

.END

Specifies to NBASM that end of file was reached. Any text after the .end directive will be ignored. The .end directive must be used for each source file.

.EVEN

Aligns the next statement/data on a word boundary. Pads with a NOP if necessary.

.EXIT

System Macro. Simply places the 'Exit to DOS' code in your file:
mov ah,4Ch
int 21h

If you specify an 8 bit value in the form of an immediate or a single quoted char, then NBASM will use this value as the AL portion of the code (the RC/ERRORLEVEL), else the AL register will be assumed:

.exit 01h	=	mov ax,4C01h
.exit 22	=	mov ax,4C16h
.exit 'A'	=	mov ax,4C41h
.exit	=	mov ah,4Ch

.EXTERNAL symbol1, symbol2, symbol3, ...

Specifies that the following symbol(s) is/are external.
COM files:

If .external is used in the TINY model (see below), then NBASM loads the correct library and adds the code to the end of the .com file for these external procedures. Please see 'Using the Library' for more info on calling external procedures in .com files.

Note: If a symbol is declared external via .external but is not used in the code, NBASM will NOT add the associated code to the end of the .com file.

Note: If the external procedure used is of a higher processor type than what has been currently declared, then an error will be returned.

If you use any .external procedures, any undeclared data will be overwritten with the external code (see DUP)

EXE files: (Currently not supported)

.IF/.ELSE/.ENDIF

NBASM allows conditional assembling. It is very easy to use. Simply include an immed7/8/16/32 as the parameter in the form of a literal constant, math equation, or as an EQUate.

If the value given is 0 (false), the following code is not assembled until an .ELSE or an .ENDIF statement is found.

If the value is not 0 (true), then the code is assembled until an .ELSE or an .ENDIF statement is found. If an .ELSE statement is found, then the rest of the code is not assembled until the .ENDIF statement is found.

```
Debug EQU 00h ; (false)
```

```
.IF Debug
; do NOT assemble this code
.ELSE
; DO assemble this code
.ENDIF
```

```
.IF Debug
; do NOT assemble this code
.ENDIF
```

```
.IF 1 ; one is considered not zero, not false.
; DO assemble this code
.ELSE
; do NOT assemble this code
.ENDIF
```

Please note that the .IF/.ELSE/.ENDIF conditional assembler directives can NOT be nested, and that NBASM does not check for a valid .IF/.ELSE/.ENDIF block. If it isn't a valid .IF/.ELSE/.ENDIF block, unexpected errors could happen.

.JUMPS (currently not working correctly) (* DO NOT USE *)
Tells NBASM to substitute a JNcc \$+3 and a JMP label for a short conditional jump that is to far away.

Example:

```
jz short There ; ** one byte to far **  
dup 128,0 ; can be any code or data  
There:
```

Gets replaced with:

```
jnz short Here ; (jnz $+3)  
jmp There  
Here: dup 128,0 ; can be any code or data  
There: ; *But why would you jump to data?*
```

You can also use the /J on the command line instead of this directive. Both are the same and both can be used together with no error.

.LIST (0 or not(0))
Tells NBASM to stop/start sending data to the .lst file. A value of 00h means stop, a non 00h value means start.

If the /X parameter was not used on the command line, the .LIST directive is ignored.

By default, NBASM sends data to the .lst file if /X was found on the command line. The .LIST directive is not needed to start the .lst file.

```
.LIST 0 ; stop sending things to the .lst file  
.LIST 1 ; start sending things to the .lst file
```

.MODEL (TINY or SMALL)
Defines the output file as either a COM file or an NBO file. If no .model directive is encountered, NBASM returns an error. If a .model directive is found, it must be before all other directives.

Example:

```
.MODEL TINY ; declares a COM file for output  
.MODEL SMALL ; declares an NBO file for output
```

.NORET (no return)
Since NBASM returns an error if you do not use a RET inside your PROC/ENDP block, you might want to use .NORET. This directive does exactly what a RET would do, except the actual opcode for RET is not added to your code. Use this directive when you want to use the USES directive so that the stack will remain clean.

.OPTOFF
Turns the optimizer off

.OPTON

Turns the optimizer on

.PAGE

Aligns the next statement/data on a page boundary (512 bytes).
Pads with NOP's if necessary.

.PARA

Aligns the next statement/data on a paragraph boundary (16 bytes)
Pads with NOP's if necessary.

.STACK

Specifies the amount to reserve for the stack. If an operand is not specified, then 256 is assumed. If an operand is specified, it must be in the range of 64 - 65534, and tells NBASM to assign that many bytes for the stack. (NBASM will increment to next even number if needed)

```
.STACK      ; default 256 bytes
.STACK 128  ; use 128 bytes
.STACK 127  ; use 128 bytes
```

.START

Specifies to NBASM to do the startup code. This directive must be before any other code, but it is not required by NBASM to assemble.

In the TINY model (COM), this directive calculates the amount of memory that the program needs, adds the stack size, (given by the .stack directive or 256 if .stack not used), and resizes the memory block to this size. Then the .start directive points SP to the top of this block.

If you use this directive in a COM file, you can not use the RET instruction to exit to DOS because SP doesn't point to offset 0FFFFh anymore. i.e.: [SP] is not guaranteed to be 00h, and doesn't point to the INT 20h vector in the PSP.

If you use the optimizer option and the .start directive, NBASM will not change INT 20h to RET.

In the SMALL model (NBO), this directive processes the segment addressing parameters ('puts the correct values in the segment registers') as well as to free the unused memory blocks. If this directive is not used in the SMALL model, you will have to write your own startup code, and segment addressing.

Example:

```
.START ; that's it, nothing else needed.
```

If you don't want to use the .START directive, use the following in your '.model small' files (see macro section at end of this file):

```
mov ax,%data
mov ds,ax
```

.x86

Allows all instructions to be assembled

Memory References

To access memory outside the program segment, you simply move a new segment address into the DS register, then address using offsets in the new segment. The memory option of the EQU pseudo-op allows you to give a variable name to offsets in other segments. For example, to access the graphics character table in ROM:

```
BIOS      EQU  F000H
CHARTABLE EQU  [FA6EH]
MOV AX,BIOS      ; can't move immed. to DS
MOV DS,AX
MOV AL,CHARTABLE ; 1st byte of char table
```

Code Branching

NBASM supports 4 instructions for branching outside the program segment.

Direct CALL and JMP

New values for the IP and CS registers are included in the instruction as two immediate operands.

Example:

```
SEGMENT1 EQU 01234h
OFFSET1  EQU 05678h
JMP Offset1,Segment1
```

Remember that the offset is the first operand, and the segment is the second operand.

Indirect CALLF/CALL FAR and JMPF/JMP FAR

Four consecutive bytes in memory are initialized with new values for the IP and CS registers. The CALLF or JMPF then references the address of the new values.

Example:

```
MOV    [DI],Offset1
MOV    [DI+2],Segment1
CALLF  [DI]
CALL  FAR [DI]
SEGMENT1    DW 1234h
OFFSET1     DW 5678h
```

Instructions

The following list is a list of all mnemonics that are allowed with NBASM and a short description. In the column before the mnemonics is the lowest allowed processor. If it is blank, then an 8086/88 is assumed. (186+ = 80186 or better, etc)

If you know of any 586 or less mnemonic/instruction not listed here, please let me know. (I am sure I am missing a few.)

(Please see notes on FPU instructions below)

	AAA	- Adjust after BCD addition
	AAD	- Adjust before BCD division
	AAM	- Adjust before BCD multiplication
	AAS	- Adjust after BCD subtraction
	ADC	- Arithmetic addition with carry
	ADCB	- (byte)
	ADCD	- (dword)
	ADCW	- (word)
	ADD	- Arithmetic addition
	ADDB	- (byte)
	ADDD	- (dword)
	ADDW	- (word)
	AND	- Logical AND
	ANDB	- (byte)
	ANDD	- (dword)
	ANDW	- (word)
286+	ARPL	- Adjust RPL field or segment selector
386+	BSF	- Bit scan forward
386+	BSR	- Bit scan reverse
486+	BSWAP	- 32bit reg from little-endian -> big-endian
386+	BT	- Bit test
386+	BTC	- Bit test and complement
386+	BTR	- Bit test and reset
386+	BTS	- Bit test and set
	CALL	- Call Procedure
	CALLF	- (far)
	CALLN	- (near)
	CBW	- convert signed byte (AL) to word (AX)
386+	CDQ	- convert signed dword (EAX) to dword pair (EDX:EAX)
	CLC	- Clear carry
	CLD	- Clear direction flag
	CLI	- Clear Interrupt flag
386+	CLTS	- Clear Task-Switched Flag in CR0
	CMC	- Complement the carry flag
686+	CMOVcc	- Conditional MOV reg, reg/mem (please note that currently NBASM does not have a .686 (Pentium Pro) directive. Use the .586 directive)
	CMP	- Compare
	CMPB	- (byte)

```

    CMPD      -      (dword)
    CMPSB    - Compare String Byte
    CMPSW    - Compare String Word
    CMPW     -      (word)
586+  CMPX8B - Compare and exchange 8 bytes
486+  CMPXCHG - Compare and exchange
486+  CPUID   - Return CPU ID using EAX in EDX:EAX
    CWD      - convert signed word (AX) to word pair (DX:AX)
386+  CWDE    - convert signed word (AX) to dword (EAX)
    DAA      - Adjust after addition
    DAS      - Adjust after subtraction
    DEC      - Decrement
    DECB     -      (byte)
    DECD     -      (dword)
    DECW     -      (word)
    DIV      - Unsigned divide
    ENTER    -
    F2XM1    -
    FABS     -
    FCHS     -
387    FCOS   -
    FDECSTP  -
    FINCSTP  -
    FLD1     -
    FLDL2E   -
    FLDL2T   -
    FLDLG2   -
    FLDLN2   -
    FLDPI    -
    FLDZ     -
    FNOP     -
    FPATAN   -
    FPREM    -
387    FPREM1 -
    FPTAN    -
    FRNDINT  -
    FSCALE   -
387    FSIN   -
387    FSINCOS -
    FSQRT    -
    FTST     -
    FWAIT    - wait for coprocessor to finish
    FXAM     -
    FXTRACT  -
    FYL2X    -
    FYL2XP1  -
    HLT      - wait for an interrupt
    ICEBP    - Interrupt trap to debugger (interrupt 1) ** Undocumented **
    IDIV     - Signed divide
    IMUL     - Signed integer multiplication
386+  IMUL   -      (reg16,reg/mem)
186+  IMUL   - Signed integer multiplication (reg16,immd8)

```

```

186+ IMUL    - Signed integer multiplication (reg16,immd16)
      IN     - Get AL/AX/EAX from port (DX)
              (DX is used if any other 16 bit register is supplied)
      INC    - Increment
      INCB   - (byte)
      INCD   - (dword)
      INCW   - (word)
186+ INSB   - In String Byte (AL)
386+ INSD   - In String DWord (EAX)
186+ INSW   - In String Word (AX)
      INT    - Software interrupt
      INT3   - Interrupt trap to debugger (interrupt 3)
      INTO   - Interrupt on Overflow
486+ INVD   - Invalidate Internal Caches
486+ INVLPG - Invalidate TLB Entry
      IRET   - Return from interrupt
386+ IRETD  - Return from interrupt (32 bit)
      JA     - Jump if above (CF=0, ZF=0)
      JAE    - Jump if above or equal (CF=0)
      JB     - Jump if below (CF=1)
      JBE    - Jump if below or equal (CF=1, ZF=1)
      JC     - Jump if carry (CF=1)
      JCXZ   - Jump if CX = 0
      JE     - Jump if equal (ZF=1)
      JG     - Jump if greater than (ZF=0 or SF=OF)
      JGE    - Jump if greater than (ZF=OF)
      JL     - Jump if less than (ZF!=OF)
      JLE    - Jump if less than or equal (ZF=1 or ZF!=OF)
      JMP    - Unconditional Jump
      JMPF   - (far)
      JMPN   - (near) (same as JMP above)
      JMPS   - (short)
      JNB    - Jump if not below (CF=0)
      JNC    - Jump if no carry (CF=0)
      JNE    - Jump if not equal (ZF=0)
      JNO    - Jump if no OverFlow (OF=0)
      JNP    - Jump if no Parity (PF=0)
      JNS    - Jump if no Sign (SF=0)
      JNZ    - Jump if no Zero (ZF=0)
      JO     - Jump if overflow (OF=1)
      JP     - Jump if Parity (PF=1)
      JPE    - Jump if Parity Even (PF=1)
      JPO    - Jump if Parity Odd (PF=0)
      JS     - Jump if Sign (SF=1)
      JZ     - Jump if Zero (ZF=1)
      LAHF   - Load into AH lower byte of flags
386+ LAR    - Load access rights
      LDS    - Load DS segment offset of
      LEA    - Load affective address
186+ LEAVE  - Leave
386+ LEAVED - Leave (32 bit)
      LES    - Load ES segment offset of

```

```

386+ LFS      - Load FS segment offset of
386P+ LGDT   - Load Global Desc Table Register
386+ LGS     - Load GS segment offset of
386P+ LIDT   - Load Interrupt Desc Table Register
386P+ LLDT   - Load Local Desc Table Register
286P+ LMSW   - Move CR0 (machine status word)
286  LOADALL - Load all registers from 0:0800h (opcode: 0Fh 05h)
3/486 LOADALL - Load all registers es:edi (opcode: 0Fh 07h)
LOCK       - Lock out other processors until finished
LODSB     - Lode String Byte (AL)
386+ LODSD   - Lode String DWord (EAX)
LODSW     - Lode String Word (AX)
LOOP      - Loop until CX = 0
LOOPE     - Loop with equal (CX >0 and ZF=1)
LOOPNE    - Loop with not equal (CX >0 and ZF=0)
LOOPNZ    - Loop while CX >0 and ZF=0
LOOPZ     - Loop while CX >0 and ZF=1
386+ LSL     - Load segment limit
386+ LSS     - Load SS segment offset of
MOV       - Move right operand into left operand
MOVB     - (byte)
MOVD     - (dword)
MOVSB    - Move String Byte
386+ MOVSD   - Move String DWord
MOVSW    - Move String Word
386+ MOVSX   - Move and sign extend
386+ MOVZX   - Move and zero extend
MOVW     - (word)
MUL      - Unsigned Multiplication (mul reg8/16)
MULB     - Unsigned Multiplication (mul byte [xxxx])
MULW     - Unsigned Multiplication (mul word [xxxx])
NEG      - Negate (two's complement; multiply by -1)
NEGB     - (byte)
NEGD     - (dword)
NEGW     - (word)
NOP      - No Operation
NOT      - Logical NOT (one's complement)
NOTB     - (byte)
NOTD     - (dword)
NOTW     - (word)
OR       - Logical OR
ORB      - (byte)
ORD      - (dword)
ORW      - (word)
OUT      - Send AL/AX/EAX to port (DX)
          (DX is used if any 16 bit register is used)
186+ OUTSB  - Out String Byte (AL)
386+ OUTSD  - Out String DWord (EAX)
186+ OUTSW  - Out String Word (AX)
POP      - Pop top of stack into operand; inc SP by 2 (pop (e)ax)
186+ POPA   - Pop all general registers (DI SI BP SP BX DX CX AX)
386+ POPAD  - Pop all extended registers (EDI ESI EBP ESP EBX EDX ECX EAX)

```

- POPB - Pop top of stack into operand; inc SP by 2
(exactly same as POPW)
- POPD - Pop top of stack into operand; inc SP by 4
(pop dword [bx] or pop dword tempvar)
- POPF - Pop top of stack into flags reg; inc SP by 2
- 386+ POPFD - Pop top of stack into extended flags reg; inc SP by 4
- POPW - Pop top of stack into operand; inc SP by 2
(pop word [bx] or pop word tempvar)
- PUSH - Push operand on to top of stack; dec SP by [2|4]
(push (e)ax)
- 186+ PUSH - Push Immed7 CONST on to top of stack; dec SP by 2
If you use PUSH BYTE 7Fh, what gets pushed is 007Fh.
If you use PUSH BYTE 80h, what gets pushed is FF80h.
The PUSH BYTE instruction is SIGNED.
immed7's are FF80h (-128d) to 007Fh (127d)
- 186+ PUSH - Push Immed8/16 on to top of stack; dec SP by 2
(push 1234h or push offset tempstr)
- 186+ PUSHA - Push all general registers (AX CX DX BX SP BP SI DI)
- 386+ PUSHAD - Push all extended registers (EAX ECX EDX EBX ESP EBP ESI EDI)
- PUSHB - Push operand on to top of stack; dec SP by 2
(exactly like PUSHW)
- PUSHD - Push immed32 on to top of stack; dec SP by 4
(push dword 01h or push dword 1234h)
- PUSHD - Push operand on to top of stack; dec SP by 4
(push dword [bx] or push dword tempvar)
- PUSHF - Push Flags onto top of stack; dec SP by 2
- 386+ PUSHFD - Push Extended Flags onto top of stack; dec SP by 4
- PUSHW - Push operand on to top of stack; dec SP by 2
(push word [bx] or push word tempvar)
- RCL - Rotate left with carry
- RCLB - (byte)
- RCLD - (dword)
- RCLW - (word)
- RCR - Rotate right with carry
- RCRB - (byte)
- RCRD - (dword)
- RCRW - (word)
- 586+ RDMSR -
- 586+ RDPMS - Read Performance Monitoring Counters (Pentium Pro+)
- 586+ RDTSC - Read Real Time Clock Stamp
- REP - Repeat (used with STOS)
- REPE - Repeat while equal (used with STOS)
- REPNE - Repeat while not equal (used with STOS)
- REPNZ - Repeat while not zero (used with STOS)
- REPZ - Repeat while zero (used with STOS)
- RET - Return from procedure
- ROL - Rotate Left
- ROLB - (byte)
- ROLD - (dword)
- ROLW - (word)
- ROR - Rotate Right
- RORB - (byte)

```

RORD      -      (dword)
RORW      -      (word)
586+ RSM    -
SAHF      - Restore from AH to lower byte of flags
SAL       - Shift Left (Adjust) (signed numbers)
SALB      -      (byte)
SALD      -      (dword)
SALC      - *Undocumented* opcode (see Appendix A)
SALW      -      (word)
SAR       - Shift Right (Adjust) (signed numbers)
SARB      -      (byte)
SARD      -      (dword)
SARW      -      (word)
SBB       - Arithmetic subtraction with borrow
SBBB      -      (byte)
SBBD      -      (dword)
SBBW      -      (word)
SCASB     - Scan String Byte
386+ SCASD - Scan String DWord
SCASW     - Scan String Word
386+ SETcc - SET byte on condition of flags
386P+ SGDT - Store Global Desc Table Register
SHL       - Shift Left
SHLB      -      (byte)
SHLD      -      (dword)
SHLW      -      (word)
SHR       - Shift Right
SHRB      -      (byte)
SHRD      -      (dword)
SHRW      -      (word)
386P+ SIDT - Store Interrupt Desc Table Register
386P+ SLDT - Store Local Desc Table Register
286P+ SMSW - Move CR0 (machine status word)
STC       - Set carry flag
STD       - Set direction flag
STI       - Set interrupt flag
STOSB     - Store String Byte
386+ STOSD - Store String DWord
STOSW     - Store String Word
386+ STR   - Store task register
SUB       - Arithmetic subtraction
SUBB      -      (byte)
SUBD      -      (dword)
SUBW      -      (word)
TEST      - Same as AND but neither operands are changed
TESTB     -      (byte)
386+ TESTD -      (dword)
TESTW     -      (word)
??? UD2   - Undefined Opcode instruction
386+ UMOV  - *Undocumented* opcode (see Appendix A)
386+ VERR  - Verify for reading
386+ VERW  - Verify for writing

```

	WAIT	- wait for coprocessor to finish
486+	WBINVD	- Write Back and INValidDate cache
586+	WRMSR	- Write to Model Specific Register
486+	XADD	- Exchange and add
	XCHG	- Exchange operands
	XLAT	- Translate from table (BX=address, AL=offset from 0 (1st))
	XLATB	- Translate from table (BX=address, AL=offset from 0 (1st)) (use XLAT cs:[BX] when you need an override)
	XOR	- Exclusive OR
	XORB	- (byte)
	XORD	- (dword)
	XORW	- (word)

FPU instructions

Some other assemblers include an [F]WAIT instruction before the actual FPU instruction if the processor directive is less than a .286 directive. Currently, NBASM does not prefix the [F]WAIT instruction. It is up to the user to prefix this instruction if s/he thinks it is needed.

Miscellaneous Items

Errors

NBASM will return any errors to the screen in the following format:

```
filename.[asm|inc](line number): error number: short description
```

Symbol space free: amount of byte(s) free

Error(s) detected: number of errors

Diagnostic(s) offered: number of 'D' errors

filename.asm is the source file that NBASM found the error. This is usually the initial source file. However, if you have included files, this could be one of the include filenames.

(linenumber) is the line number of the 'filename.asm' that NBASM found the error.

error number is the number of the error found.

short description is just that (see below for a detailed description).

Symbol space free will display the amount of bytes left in the symbol table.

Error(s) detected is the number of critical errors.

Diagnostic(s) offered is the number of items that could be changed to make the source 'better'. Most of the time it will say things like:

'Use short jmp' - use short rather than long

See Running NBASM for errors returned in ERRORLEVEL

Detailed description of assemble time errors:

0Ah = 'Error opening Lib File'

NBASM can not find the library file wanted. The lib file should either be in the current directory or pointed to by the LIB= environment string

0Bh = 'Error reading from Lib File'

The wanted library file is either corrupt or not a valid NBASM lib file.

0Ch = 'Include file not found'

The specified included file could not be found. The inc file should either be in the current directory, pointed to by the INCLUDE= environment

string,

or specified with the /I{} command line parameter.

0Dh = 'Include filename needed'

Specify a filename after the include keyword

0Eh-11h not specified

12h = 'Unexpected end of line'

Most of the time, NBASM has found a string that doesn't have the closing mark of a single or double quote.

13h-17h not specified

18h = 'End of file before .END'

All source files, including include files need to have a line with the

.end

as the last assembled line.

19h not specified

1Ah = 'Segment near (or at) 64k limit'

Most of the time this means you are trying to create a .COM file that is larger than 64k.

1Bh = 'Block nesting error'
Used with in INCLUDE file nesting. You can only have ten (10) include files nested at once.

1Ch-1Dh not specified

1Eh = 'Redefinition of symbol'
You have a symbol with the same name as a previously defined symbol. Remember that all NBASM symbols are NOT case sensitive.

1Fh not specified

20h = 'Phase error between passes (+|-difference)'
(prints this message on the first occurrence only)

21h = 'Symbol not defined'
NBASM has found a symbol that you have not defined. Check your spelling of the symbol

22h = 'Syntax error'
Is a general syntax error.

23h = 'Type illegal in context'
You have tried to use an operand that is either the wrong size or most likely the wrong type.

24h-27h not specified

28h = 'forward reference illegal'
You have used a forward reference in an EQUate line. All EQUates must be constant or previously declared symbols.

29h-2Dh not specified

28h = 'operand expected'

2Fh-35h not specified

36h = 'only one operand allowed'
You have tried to put too many symbols in an expression.

37h not specified

38h = 'operand expected'

39h-41h not specified

42h = 'improper operand type'
Mostly used in DB and DW lines.

43h = 'jump out of range by ????? bytes'
Specifies that you have used a short jump where a long one is needed.

44h-52h not specified

53h = 'for use in .COM files only'

54h = 'forward reference needs override or far'
Used with PROC FAR

55h = 'illegal value for DUP count'
Supply a smaller dup count

56h not specified

57h = 'PROC nesting too deep'
Only ten (10) nesting levels

58h-59h not specified

5Ah = '.START must be at 100h for .COM files'
.start must be before any other instructions

5Bh not specified

5Ch = 'Undefined DUP used before this location'
You have used an Undefined DUP (dup count,?) line before this location

and
the data is not in phase

5Dh-5Fh not specified

60h = 'wrong length for override value'
Most likely used the wrong segment register

61h not specified

62h = 'Forward direction only'
Only use a forward direction with ORG

63h-69h not specified

6Ah = 'Open PROC (needed ENDP)'
Need to supply an ENDP with every PROC

6Bh-72h not specified

73h = 'Too many user symbols (1100 max (13 bytes each))'
You have used up the symbol space.

74h = 'Diagnostic: Specify (d)word or byte operation'
Need size: Byte or Word or Dword specification

75h = 'ENDP without PROC'
Found ENDP without a valid PROC. NBASM Probably found error with PROC
line.

76h = 'Diagnostic: Could use JMPS or JMP SHORT'
Simply change a long jump to a short jump by adding the short keyword.
(jmp short there)

77h = 'NEAR or FAR expected'
Need a type specifier with the PROC keyword

78h = 'Data too long'
Most likely you are putting an immed16 into a DB

79h = 'Illegal size for stack'
Stack size can only be 64 bytes to 65534 bytes long.

7Ah = 'Not allowed in COM file'
You have used a function that can not be used in a .COM file.
- Can not use .DATA
- Other items specified in this documentation

7Bh = 'Symbol not found in Lib File'
You have tried to call an external proc not found in the specified
library.

7Ch = 'Need .MODEL [type]'
Specify .tiny or .small

7Dh = 'Symbol needs 186'

7Eh = 'Symbol needs 286'

7Fh = 'Symbol needs 286P'

80h = 'Symbol needs 386'

81h = 'Symbol needs 386P'

82h = 'Symbol needs 486'

83h = 'Symbol needs 586+ (Pentium/Pro/II/III)'

84h = 'Symbol needs .87'

85h = 'Symbol needs .187'

86h = 'Symbol needs .287'

87h = 'Symbol needs .387'

88h not specified

89h = 'Only read first 65535 bytes of include file'
Include files can be only 65535 bytes in size

8Ah = 'Can not use ret as 'exit code' in .nbo'
You must use .exit or define your own exit code

8Bh = 'Can not use ret if '.start' used'
The .start system macro changes the Stack segment.
Therefore you can not use the PSP:[00] ret trick.

8Ch = 'Error with Lib File'

The wanted library file is either corrupt or not a valid NBASM lib file.

8Dh-FEh not specified

FFh = 'Unknown error'

Precautions:

Please note that all code that is used/called from a library with the .external procname technique is added to the end of your .com file. So if you plan to use the remaining memory after your .com file, please take in mind the size of the code included by the library, or you might overwrite the code and crash the machine.

I plan to add some code and documentation to NBASM to allow the coder to find where the end of this added code will be. Until then, be careful about assuming where the code is and how long it is.

I have added most of the 32 bit instructions and registers. However, I have not fully tested them yet, so please use caution when using the 32 bit instructions. I will fully test them soon.

Examples

(See the included DEMO1.ASM file for another example)

-- Example #1: -----

; this example shows the use of the '@LABEL' directive.

.model tiny

.code

```

mov ah,09          ; print string
mov dx,offset @label ; use 'unique' name
int 21h
mov ah,09          ; print string once more
mov dx,offset @label ; use same 'unique' name
int 21h
jmp short SkipData

```

```
@label db 'Both of the @label above points to this place (1)',36
```

```
SkipData mov ah,09          ; print string again
```

```

mov dx,offset @label ; use same 'unique' name
int 21h
int 20h              ; exit

```

```
@label db 'The above @label points to this place (2)',36
```

.end

-- Example #2: -----

.model tiny ; create COM file

.186 ; allow 186 instructions

.external prthex, prtstring ; include the code for prthex and prtstring

.code ; start of code segment

```

push offset msg1 ; use library function (prtstring)
call prtstring ;
mov ax,offset msg2 ; use library function (prtstring)
push ax ;
call prtstring ;
push 1234h ; put 1234h on the stack
call prthex ; and print it as hex
mov dl,'h' ; print the trailing 'h'
mov ah,02 ;
int 21h ;
int 20h ; exit to DOS

```

```
msg1 db 13,10,'This is an example of the .external directive',0
```

```
msg2 db 13,10,"For this example we use the 'prtstring' "
```

```
db "and 'prthex' routines"
```

```
db 13,10,'We will print the value 1234h here: ',0
```

.end

Appendix A - Undocumented instructions supported by NBASM

** SALC **

Instruction: SALC (accepts no arguments)
Name: "Set AL to Carry"
opcode: D6h
Min Processor: .8086
Max Processor: none
Flags modified: none

This instruction sets AL to 0FFh if the Carry Flag is set (CF=1), or clears AL (00h) if the Carry Flag is clear (CF=0).

** POP CS **

Instruction: POP CS
Name: "POP CS"
opcode: 0Fh
Min Processor: .8086
Max Processor: .286
Flags modified: none

Moves the current word from the stack into CS.
***** Only works on the 8088/8086 through the 80286.
From the 80386, Intel used this opcode as an extension opcode.

** UMOV **

Instruction: UMOV
Name: "User MOVE"
opcode: 10h-13h
Min Processor: .386
Max Processor: .486
Flags modified: none

This instruction is identical to the following instructions:
MOV memreg,reg
MOV reg,regmem

The only difference is that when the CPU is in the state which it has the hidden memory active, the user can now transfer data to and from the user memory while the hidden memory is active.

I have not tested this instruction, so use CAUTION when using this instruction. Make sure that NBASM assembled it correctly before you use it. If you have tested this instruction, please let me know what you find. Thank you

Appendix B - Write your Boot Sector to disk using NBASM

NBASM will write your binary image to the boot sector of the specified disk if you include the /w<d> parameter, where <d> is either a or b. (remember that the 'w' as well as the 'a' and the 'b' are to be lowercase letters)

NBASM first makes a few tests on your binary image. They are:

- your code must start with the JMP (E9h) or JMP SHORT (EBh) opcode.
- your code must be 512 bytes in length
- the word at offset 01FEh must be: 0AA55h

if NBASM finds any of these to be in error, it will not write the image to the disk, and will return an error.

To write your image to drive a:, use the following command line:

```
NBASM boot boot.bin /wa
```

To write your image to drive b:, use the following command line:

```
NBASM boot boot.bin /wb
```

NBASM will not allow any other drive letter.

The 'boot.bin' in the two examples above are not needed to write your image to the boot sector of the disk. However, NBASM creates an image in the current directory just as if you hadn't used the /w<d> parameter. This way you can name it using 'boot.bin' as I did above.

Please use this parameter with caution. If your Boot Sector Image has any errors in it, the disk will not be bootable until it is written to again, with a correct/working boot sector.

Please make sure the disk is in the drive before NBASM assembles your code or an error will return.

Structures and Struct

NBASM allows C like structure use. You give the struct a name and use the STRUCT declaration keyword. Then you must declare each member with an appropriate size of: byte, word, dword, fword, qword, or dup.

Example:

```
STRUCT_NAME  struct
member1      byte          ; member1 is of byte size
member2      word          ; member2 is of word size
member3      dword         ; member3 is of dword size
member4      fword         ; member4 is of fword size (6 bytes)
member5      qword         ; member5 is of qword size (8 bytes)
member6      dup 10        ; member6 is 10 bytes in length.
ends         ; end of structure declaration.
```

A Note: Make sure that ENDS does not start in column one.

NBASM places this structure in the Symbol Table. However, your code does not have access to it yet. You must define a symbol with the type of this structure using the ST directive. Example:

```
OurStruct  st  STRUCT_NAME
```

Now you can access the structure and its members.

```
mov  ax,OurStruct.member2
mov  eax,OurStruct.member3
```

Please note, as with C, you can define as many symbols with this type as you have room. However, you must define the structure type before you use the ST directive to declare the symbol.

```
OurStruct  st  STRUCT_NAME
NewStruct  st  STRUCT_NAME
Astruct    st  STRUCT_NAME
```

All three above are valid symbols, are of type STRUCT_NAME, and occupy three different memory blocks.

You can also initialize data to the structure when you use the ST directive. Look at the following line:

```
OurStruct  st  STRUCT_NAME uses 12h,1234h,12345678h,12345678h,12345678h,"ab"
```

The above line would create the symbol OurStruct of type STRUCT_NAME and initialize each of its members to the values that follow the USES directive respectively.

***** Please note: Currently you can only place the initialization data on the SAME line as the declaration. Also, if you want to initialize one of the members, you must initialize all members before that member. You do not have to initialize all members after the last desired initialization.

A note about using the 'DUP immed' declaration. You place a value after the DUP directive to tell NBASM what size to create this member. If you want to declare a string to this member, you must declare the size large enough to hold the proceeding NULL byte (if any). If you declare a member as:

```
member6 dup 10 uses "0123456789"
```

The string will not be null terminated.

However, if you do the following:

```
member6 dup 11 uses "0123456789"
```

NBASM will null terminate the string for you. If the size of the member plus the NULL terminator is larger than the declaration size, the rest of the data in the member's space is undefined and will be random values.

For example:

```
member6 dup 10 uses "01"
```

Will declare a member with the following string value:

```
30h 31h 00h ??h ??h ??h ??h ??h ??h ??h
```

You may use the SIZEOF directive to get the size of a structure by using the following:

```
mov ax, sizeof STRUCT_NAME
```

Notice that you MUST use the TYPE define rather than the actual symbol you declare with the ST directive. To see more on SIZEOF, go to page 53.

NBASM allows the C like SIZEOF directive. The following datatypes are currently available for use with the SIZEOF directive.

1. byte, word, dword, fword, qword
2. STRUCT declarations

**** I hope to expand to more types soon ****

Example of the types listed in number 1 above:

```
mov ax, sizeof byte ; returns ax = 1
mov ax, sizeof word ; returns ax = 2
mov ax, sizeof dword ; returns ax = 4
mov ax, sizeof fword ; returns ax = 6
mov ax, sizeof qword ; returns ax = 8
```

Example of the type listed in number 2 above:

```
newstruct struct
member1 byte
member2 dup 10
member1 dword
ends

mov ax, sizeof newstruct ; returns ax = 15
```

*** Please note that the SIZEOF directive is not complete. I hope to add more functionality to it soon. A few things I plan to add: size of actual symbols, string lengths, immediates, etc.