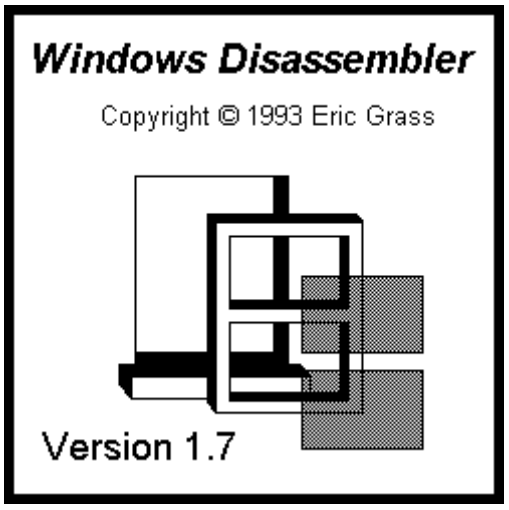
Windows Disassembler 1.7b

A 486 Disassembler for Windows



User's Manual

Note: Version 1.07b contains bug fixes for the incorrect disassembly of certain 387/486 floating-point instructions in version 1.07. For more information, see the section entitled **Differences Between Versions**.

Index

Introduction

Windows Disassembler disassembes Windows executables and dynamic link libraries. It allows the user to browse at the source code of a program without having to write it to a file. *Windows Disassembler* generates procedure directives, as well as all of the literal Windows API function call names.

Specifications

Files
Works on Windows 3.*x* executables and dynamic link libraries only.
Instruction Set
Translates all instructions within the 486 instruction set. It assumes that all code is executed in 16-bit mode (since Windows 3.1 uses 16-bit mode only).
Operating System and Hardware
Requires at least DOS 4.0, Windows 3.1, and a 286 or above IBM compatible computer. Installation of *SMARTDRV* (which comes with Windows) is recommended.

Operation

Opening Files

The default file name extension is "**.exe**" for opening files if no extension is specified. *Windows Disassembler* processes one file at a time. If a file is opened while another one is already open, the old file will be automatically closed. When opened, the file's assembly language code appears on the screen, provided that the file has a DOS executable file header, a new executable file header, and at least one segment. Otherwise, a dialog box will inform the user that the file does not meet a particular specification.

The Display

Displaying code in the display window is presented as an alternative to generating a gigantic assembly language source code file, since some programs are large, and the user may merely want to glance at a program's source code.

The code that initially appears in the window when a file is opened is the first segment within the file. Numbers are assigned to segments according to their chronological order within the new executable file header. *Windows Disassembler* displays one segment at a time within the window. The **View | Segment** command must be used to go to another segment. To scroll the text in the window, use the Up Arrow, Down Arrow, Page Up, and Page Down keys, or the scroll bar. To see the address offsets of each instruction, select **View | Address Offsets** from the main menu. To jump to a specific address, select **View | Go To** from the main menu and enter the address in hexadecimal format.

The **View | Far Call Names** command toggles between displaying far function call names and the actual relocation values in far **CALL** instructions (for example, **0000H:0FFFFH**).

All labels have the form of either **LxxxxH** or **DxxxxH**, where **xxxx** is a 4-digit hexadecimal number equal to the offset of the location being referenced. Labels with an **'L'** prefix denote locations within the immediate code segment, and labels with a **'D'** prefix denote locations within a data segment. Labels within a code segment can either be procedure labels, jump/loop labels, or data labels within the code segment. Assembler directives, while generated for source code text files, are not shown in the display window.

Strings are detected and translated by *Windows Disassembler* whenever five or more visible characters occur within a data segment.

The **Edit!** command allows the user to convert a desired range of bytes from byte declarations into instructions, or vice versa, or to give labels to a specified range of bytes. This command is necessary for programs which have data declarations in their code segments. Note that all modifications which the user has made to a segment will be lost when exiting that segment. The user can save that segment using the **Save Current Segment Only** option as a text file first before quitting to save the changes. However, when the user leaves the segment, there is no way to restore the byte settings except by specifying them over again. Selecting the **Create Separate Files For Each Segment** option will result in the the modifications/settings being erased (lost) *before* the file is created, hence the user must use the **Save Current Segment Only** option.

Creating Assembly Language Source Code Files

After opening an executable, the user can create an assembly language source code file for it using the **Save Text As** command. If the source code file name that the user specifies is the name of an already existing file, then that file will be automatically overwritten with the new source code file. Three options are available for generating (a) file(s). The first is to put all of the source code into one file. The name of this file will be the name the user specifies. The second option is to put each segment of the source code into separate files. Each segment's file name will be of the form ***yournameN.ext***, where ***yourname.ext*** is the name the user specifies in the dialog box, and ***N*** is an integer corresponding to the segment's number and which is appended to the base-name of the file (if necessary, this base name will be truncated to perform the appending). For example, if the user specifies **\work\myprog.asm** as the file name, *Windows Disassembler* will generate files named **\work\myprog1.asm**, **\work\myprog2.asm**, **\work\myprog3.asm**, etc.. The third option is to generate a file for the current segment only (which is currently being displayed in the window). In this case *Windows Disassembler* uses the file name exactly as specified.

All editing done will be lost if the user exits a segment which the user has just modified, or if the user tries writing all of the segments to a file(s) at one time. However, if the user uses the **Save Current Segment Only** option, all modifications will remain.

The new file will contain tabs. To display the file in the way in which it was intended to be displayed, the user should set his or her editor's tab stop option to 8 spaces.

Windows Disassembler will create **TITLE**, **.CODE *segmentname***, **.DATA *segmentname***, **.MODEL LARGE**, **.486**, and **EXTRN *winAPIfunc*:FAR** directives. **PROC** and **ENDP** directives are also created for all exported and far procedures. In the case of non-exported functions, these procedure directives will all have the following form:

```
Function n    PROC  FAR PUBLIC
              (code)
              RETF
Function n    ENDP
```

where ***n*** is the ordinal number (a decimal integer value) of the procedure in the entry table of the program's executable file header. For exported functions, the name of the function is explicitly written as it is listed in the resident and non-resident names tables in the program's header. For calls to functions in fixed memory segments, a comment is written beside the call. For example,

```
CALL   FAR PTR Proc0AD0HSeg5      ; (Fixed Memory Location)
```

For far calls to procedures within the program in a different segment, **EXTERNDEF**'s are generated. Near procedures are written in the following form:

```
ProcXXXXSegN      PROC  FAR PUBLIC
                  (code)
                  RET
ProcXXXXSegN      ENDP
```

where ***XXXX*** is a four-digit hexidecimal value equal to the offset of the procedure in the segment and ***N*** is the decimal number of the segment the procedure is in.

Windows Disassembler generates segment names for segment directives of the form **.CODE SEG*n***, where ***n*** is the segment number. This name is produced in order to distinguish between segments, and can be deleted or changed. (If the segments are in separate files then the name isn't needed.) If there are exactly 2 segments in a program, *Windows Disassembler* treats the program as having a small model, otherwise it assumes the program has a medium memory model. If the program has a compact or large model, then the **MODEL** directive must be changed to reflect the actual memory model. *Windows Dissassembler* 1.7b translates functions belonging to **commdlg.dll** and **shell.dll**. It also generates information for unknown function calls in the form **Module *modulename* Ordinal *n***. The user can look up the names of these function names using an executable-file header utility on the given dynamic link library. (In other words,

one can use the relocation table names and offsets provided by an **.exe** file header utility to determine the function/variable names in the source code.)

Finally, **EXTRN**'s (or **EXTERNDEF**'s) must be supplied for any far variables used by the program not already supplied by *Windows Disassembler* (typically the far variable **__winflags** is used by Windows programs, for example).

As an example, the files **hello.exe**, **hello.c**, **hello.def**, **hello2.inc**, **hello1.asm**, and **hello2.asm** are included to demonstrate disassembly using *Windows Disassembler*. **hello.exe** (a "hello world" program) is a compilation of **hello.c**. **hello.exh** is an **.exe** file-header listing for **hello.exe** generated by *EXEHDR*.

hello1.asm and **hello2.asm** were generated using *Windows Disassembler* (using the **Create Separate Files** option) and were edited as follows. The labels **L0627H**, **L01ACH**, and **L0360H** were made global labels via the **::** (double colon) since these are accessed outside of the procedure in which they exist. (In *MASM* 5.1 the **::**'s wouldn't be necessary.) An **EXTRN __winflags** directive was added, and the segment names **SEG1** and **SEG2** were deleted.

The include file was created by copying the file **hello2.asm** to **hello2.inc**. Then, using an editor with a regular expression search function, each occurance of **"^D"** was replaced with **EXTERNDEF D**, each occurance of **DB 00[A-F,0-9][A-F,0-9]H** was replaced with **:BYTE**, and each occurance of **DB "[A-Z,a-z,0-9,\\,\.,\,\ ,*,\%,\~,\<,\>, +,=,-,?,@,_]*"** was replaced with **:BYTE**. The **EXTERNDEF**s serve as either **PUBLIC** or **EXTRN** specifiers, depending on whether the corresponding argument of an **EXTERNDEF** is located in the same file or else in a different module (like function prototypes in C).

One can rebuild **hello.exe** from **hello2.asm** with *MASM 6.0* by typing:

```
ml /c hello1.asm
ml /c hello2.asm
link /ALIGN:4 hello1 hello2,hello2,, libw slibcew, hello.def;
```

which will generate **hello2.exe**.

Borland's *Resource Workshop* can be used for obtaining the resources from executables if necessary.

Assembly Tips

A problem that normally occurs is undefined label errors because of references to labels that are located in a different procedure. The **::** operator must be used to make such labels global. Another problem is a linking error in which a given module references a global variable that doesn't exist. The problem is usually that the variable is a string which follows another non-null terminating string in the data segment and the two strings are thus combined as one string. In this case you must separate the strings. The error, "**A2006 : undefined symbol**" will occur when there are fixed relocations in the program, which require **EXTRN**s and **PUBLIC**s. However it is possible that procedure names could conflict, requiring the procedure(s) to be renamed, especially in the case of procedures with the name, **Procedure0000**.

To make the code modifiable and more readable, it is necessary that the user changes all literal addresses in the code (hexidecimal numbers) into their symbolic equivalents. For example, in the **hello** program,

```
MOV   AX, 00B0H
MOV   DX, DS
PUSH  DX
PUSH  AX
```

should be changed to

```
MOV   AX, OFFSET D00B0H
MOV   DX, DS
PUSH  DX
PUSH  AX
```

since this portion of code is passing the address of a string to a Windows function.

It is advisable that the user also makes a hardcopy of the **windows.h** file and that the user converts the

windows.h file into its *MASM* equivalent using the *H2INC* which comes with *MASM 6.0*. *H2INC* cannot translate certain macros, such as **RGB** and **MAKEINTRESOURCE**, and hence these must be manually rewritten in *MASM* or else deleted. This way, certain constants such as message values can be replaced by their symbolic equivalents. It is also suggested that the user incorporate the **prologue.inc** file which comes with *MASM 6.0* into the program in place of the existing prologue and epilogue code to make things more legible. Finally, the user should replace all other variable names and constants with more meaningful expressions. With the **windows.inc** file generated by *H2INC*, procedure calls usually can be written in a more legible form using **INVOKE**s. If the **NOCASEMAP** option is used (for employing case sensitivity), the **prologue.inc** file will need to modified slightly. In particular, the case of three or four of the words in the **prologue.inc** file will have to be changed in order to agree. **.IF**, **.WHILE**, and **.REPEAT** constructs can also be used to make the code more clear. The steps mentioned above can be accomplished faster with the help of the *HiLevel* utility.

Windows Disassembler 1.7b always outputs the **.486** directive following the **TITLE** directive in every file.

Differences Between Versions 1.5, 1.6, 1.7, & 1.7b

The display window for version 1.6 was redesigned to fit more text in the window by minimizing the space between lines.

Starting with version 1.6 procedure names have the form of **Proc*XXXX*Seg*N*** instead of **Procedure*XXXX*** in order to guarantee unique procedure names throughout the entire program.

A bug was fixed from version 1.5 which made the program go into an infinite loop in a particular (rare) situation.

The include file (**hilevel.inc**) used by the *HiLevel* utility was updated for version 1.6 to include prologue/epilogue macros and the program *HiLevel* was upgraded to accomodate the include file.

Versions 1.0 through 1.6 disassembled only 286 instructions. Version 1.7 disassembles all 486 instructions, including the floating-point instruction set.

Version 1.7b contains bug fixes for the incorrect disassembly of certain floating-point instructions in version 1.7. These bug fixes include fixes for the following bugs in version 1.7:

1.) Incorrect stack registers were supplied for instructions having one of the following stack registers as their operands: **ST(1)** through **ST(7)**.
2.) The no-wait instructions **FNCLEX**, **FNDISI**, **FNENI**, **FNINIT**, **FNSAVE**, **FNSTCW**, **FNSTENV**, and **FNSTSW** were each incorrectly translated into their corresponding wait versions. In version 1.7b, the wait version of each of these instructions is given as a **WAIT** instruction followed by the corresponding no-wait version of the instrucion.
3.) The instruction **FCOM *mem64*** was incorrectly translated as **FIDIVR *mem64***.
4.) For the instructions **FLDENV**, **FRSTOR**, **FSAVE**, and **FSTENV**, version 1.7 failed to differentiate between the 16-bit versions and the 32-bit versions of the instructions. Version 1.7b will append either a **W** or a **D** (as required by *MASM* 6.0) to these instructions in order to differentiate between the 16-bit versions and the 32-bit versions of each instruction.
5.) The instruction **FYL2X** was incorrectly translated as **FYL2XP1**.
6.) The instructions **IRET** and **IRETD** now have an **F** appended to them (i.e., **IRETF** and **IRETDF**) as required by *MASM* 6.0 to prevent epilogue code from being generated.
7.) For the 386/486 instruction **J*cond disp(2)*** (conditional near jump), the label was incorrectly calculated as **L*(xxxx-1)*H** instead of as **L*xxxx*H** (i.e., the numeric portion of the label was off by 1)

The *HiLevel* Utility

IMPORTANT: *HiLevel* 2.1 was designed to work with 286 code. If there are 386/486-specific instructions in your source code, they may cause HiLevel to halt and report a syntax error.

The *HiLevel* utility included with *Windows Disassembler* is a *Windows 3.1* utility which attempts to build high-level constructs out of the bare instructions generated by *Windows Disassembler*. The result is a smaller, more understandable, and more readily modifiable source code file. It will accept as input basic *MASM* programs, provided they do not have macros or certain other directives and high-level syntax keywords. It should accept all source code generated

by *Windows Disassembler*. *HiLevel* can construct nested **.IF** statements for each corresponding block of instructions found in the given *MASM* source code file. Locals are given symbols of the form **local*n*** and parameters are given the symbol **par*n***, where ***n*** is the offset of the variable relative to the **BP** register.

HiLevel also constructs "pseudo-function calls" via a macro procedure named **hCall**. The **hCall** macro is defined in the **hilevel.inc** which is included with *Windows Disassembler*. This macro does not perform any high-level operation, but rather is just a more legible way of performing a series of pushes followed by a procedure call, regardless of whether the arguments being pushed are actually being passed to the given function or not. *HiLevel* generates an **OFFSET D*xxxx*H** instead of ***xxxx*H** when a number ***xxxx*H** follows **DS** in the parameter list of a **hCall** invokation, since this combination is practically always a far address being passed as an argument.

The **PROC** directives produced by *HiLevel* are designed to work with either the **hilevel.inc** file or the **prologue.inc** file that comes with *MASM 6.0*. As mentioned before, when enabling case-sensitivity (via **OPTION CASEMAP:NONE**), some of the names in **prologue.inc** need to be modified in order to be made to have the same case, plus there is a defective echo statement in it which should be fixed. If *HiLevel* detects prologue code in a procedure, it then checks for matching epilogue code. If the prologue and epilogue do not logically agree, *HiLevel* generates a comment above the procedure that explains what is missing in the epilogue code, and consequently the procedure is left as is with no prologue/epilogue directives. If the epilogue and prologue logically agree, then the literal code is replaced by the appropriate prologue/epilogue directives, including the **FORCEFRAME** and **LOCAL** directives, plus by specifying any parameters.

If there is a syntax error in the source file, *HiLevel* will halt and give the line number on which the syntax error was found. Otherwise it displays the message, "**Compilation was successful! Hurrah! Hurrah!**" It may take as much as a minute to process a source code file, and as long as the user sees the disk drive light come on at regular intervals (say every 5 seconds) there is no cause for alarm. Otherwise, the system is probably hung. It is possible that *HiLevel* could hang up the system because of its limited local heap of 47,260 bytes (which is not a major problem in 386 enhanced mode, since pressing **Enter** will terminate the application. Otherwise, in standard mode, hitting **Ctrl-C** instead of **Ctrl-Alt-Delete** will sometimes terminate the application). What this means is that for programs containing extremely large procedures *HiLevel* will use up the local heap and go into outer space (stop responding). But for typical files, it should work.

As an example, the file **hellohil.asm** has been included, which is generated from **hello1.asm**. **Hellohil.asm** was assembled and linked with the old **hello2.obj** and **hello.def** files as follows:

```
ml /c hellohil.asm
link /ALIGN:2 hellohil hello2,hellohil,, libw slibcew, hello.def;
```

The only changes made were the renaming of **Proc042ASeg1** to **_aNchkstk** (because the prologue/epilogue code requires this), the addition of double colons (**::**) for the global labels, and carriage returns (lines) inserted after the labels following the **PROC** directive in procedures **Proc03EBSeg1** and **Proc03FASeg1**. The last change is because of a bug (or undocumented behavior) in MASM that requires this. The bug is that whenever a macro call or loop-generating directive (for example, **hCall**s**, .IF**s, **.WHILE**s, etc.) occurs on the line following a **PROC** directive in which prologue code is expanded, and there is no **LOCAL** directive, *MASM* mistakedly will suppose that the macro will expand into the **LOCAL** directive. When it discovers that the **LOCAL** isn't there, it just continues assembling, but consequently it somehow distorts the expansion of the macro, so that either an error is generated or else garbage instructions are generated. The following listing shows what happens if we change the **PUSH WORD PTR par4** and **CALL FAR PTR LocalFree** instructions (in procedure **Proc08A2Seg1** in **hellohil.asm**) into a **hCall** macro call:

```
089D                    Proc08A2Seg1    PROC  NEAR C <NOLOADDS, NOINCBP, FORCEFRAME,
NOCHECKSTACK>, par4:WORD
                                      hCall <FAR PTR LocalFree, WORD PTR par4>
= 0001                          1                   ??012D = 1
= FAR PTR LocalFree             2                           ??012E TEXTEQU <FAR PTR
LocalFree>
= 0000                          2                           ??012D = 0
= 0000                          3
```

```
= 0000                          3            ??0130 = 0
= 0000                          3            ??0131 = 0
= 0000                          3            ??0132 = 0
= 0000                          4         IFIDN <NOLOADDS>, <NOLOADDS>
= 0000                          4         IFIDN <NOINCBP>, <NOINCBP>
= 0001                          4         IFIDN <FORCEFRAME>, <FORCEFRAME>
= 0000                          4         IFIDN <NOCHECKSTACK>, <NOCHECKSTACK>
= FFFFFFFF                      3
089D  55                        3
089E  8B EC                     3            PUSH  BP
08A0  FF 76 04                  2            EXITM <00H>
08A3  9A ---- 0000 E            1                  CALL  ??012E
                                          RET
= 0000                          1            ??0134 = 0
= 0000                          1            ??0135 = 0
= 0000                          1            ??0136 = 0
= 0000                          1            ??0137 = 0
= 0000                          2          ??0135 = 0
= 0000                          2          ??0137 = 0
= 0001                          2          ??0134 = 1
= 0000                          2          ??0136 = 0
= FFFFFFFF              1        ??0134 = ??0134 OR ??0137 OR ??0135 OR ??0136 OR (00H NE 0)
OR
08A8  8B E5             1               MOV   SP,BP
08AA  5D                1               POP   BP
08AB  C3                1               RET
08AC                     Proc08A2Seg1       ENDP
```

Instead of **PUSH BP** and **MOV BP, SP**, it generates just **PUSH BP** in the prologue, and the push specified in the **hCall** call doesn't get expanded. Consequently, in *MASM 6.0*, you may want to avoid calling a macro or using a loop-generating directive as the first instruction in a procedure when:

a.) the automatic prologue code is forced (via **FORCEFRAME**) and
b.) there is no **LOCAL** directive.

Bugs

Known Bugs In Version 1.7b

The screen will need refreshing after scrolling upwards, primarily within data segments, but sometimes in code segments if the user edits the bytes. This bug will not affect file generation.

The scroll bar does not work properly when displaying segments of size 7FFFH or greater. In this case the user must use the **Page Up/Page Down** and the up arrow/down arrow keys. This is because of *Windows'* scroll bar range limit of 32,726 (7FFFH).

There is an occasional bug associated with references to procedures in fixed segments (as opposed to moveable segments). One such case is where the segment and offset of a function are being referenced. The user might, for example, see something like the following:

```
PUSH   SEG ABOUTDLG
PUSH   00A4H                ; (Fixed Memory Location)
PUSH   WORD PTR D0AC0H
CALL   FAR PTR MakeProcInstance
```

This would be an error. Supposing **ABOUTDLG** is in segment 1, the second **PUSH** should actually be, "**PUSH OFFSET Proc00A4Seg1**" The cause for this error has not been specifically determined.

License / Warranty Disclaimer

You are free to distribute *Windows Disassembler 1.7b* provided that no fee is charged for use, copying or distribution, it is not modified in any way, and this documentation file (unmodified) accompanies all copies. This program is provided as is without any warranty, expressed or implied, including but not limited to fitness for a particular purpose. *Windows Disassembler* may **not** be used in any unlawful or illegal manner.

Windows Disassembler 1.7b is shareware. Continued use of *Windows Disassembler* 1.7b beyond a 30-day trial period without registering is prohibited.

REGISTRATION FORM

The single license fee for *Windows Disassembler* version 1.7b is only $20.00. Registering this software entitles you to receive the latest information regarding upgrades and upgrade discounts (upgrades to minor revisions are free for registered users.) Please fill out this form (or a reasonable facsimile thereof) and send it with your check for $20.00 to the address below:

Eric Grass (314) 928-7803
1612 Gettysburg Landing
St. Charles, MO 63303

Date_____

Name_____ Phone _____

Address_____

City_____ State _____ Zip _____

Please indicate which type of disk you use: _____ 5.25" _____ 3.5"

Product: *Windows Disassembler Version 1.7b*
Total Price: $20.00 per copy
Number of copies: _____ copies x $20.00 = _____ Total Cost

Please make your check payable to Eric Grass.

Copyright

Inquiries, comments, and suggestions regarding *Windows Disassembler* 1.7 are welcomed and can sent to Eric Grass via the following:

Eric Grass
1612 Gettysburg Landing
St. Charles, MO 63303

Internet: **s876795@umslvma.umsl.edu** or **s876795@slvaxa.umsl.edu**