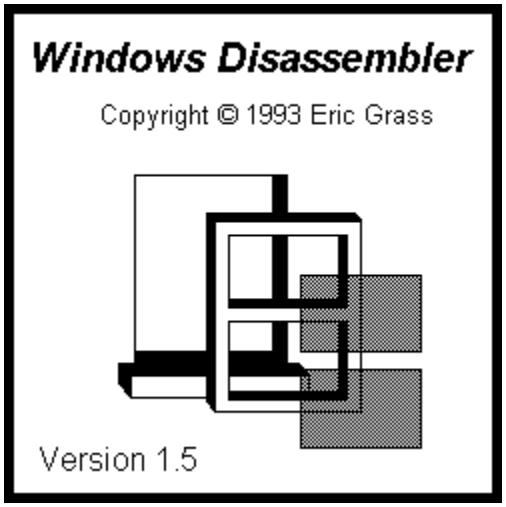
Windows Disassembler 1.5



User's Manual

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 286 instruction set with the exception of the following multi-tasking instructions: LAR, LGDT, LIDT, LLDT, LMSW, LSL, LTR, SGDT, SIDT, SLDT, SMSW, STR, VERR, and VERW.
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 **Set Byte** 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 their editor's tab stop option to 8 spaces.

Windows Disassembler will create **TITLE**, **.CODE *segmentname***, **.DATA *segmentname***, **.MODEL LARGE**, **.286**, 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 fixed functions, a comment is written beside the call indicating which segment the function belongs to. For example,

```
CALL   FAR PTR Procedure0AD0H ; (Located in Segment 5)
```

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

```
Procedure XXXX    PROC  FAR PUBLIC
                  (code)
                  RET
Procedure XXXX    ENDP
```

where ***XXXX*** is a four-digit hexidecimal value equal to the offset of the procedure within the segment.

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* 1.5 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**, **hello.exh**, **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 **hello.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**.

Make the data segment accessible to all modules by copying the contents of the data segment file to a new file and converting it into an include file. This is done (string declarations might need to be replaced manually) and then saving the file with an **.inc** extension. Then include this file (i.e., **INCLUDE** ***filebasename*.inc**) in each module that accesses the data segment. (If there are two data segments, then there could be conflicting labels.) Finally, assuming one has the resource files, assemble each module and link. Otherwise, Borland's *Resource Workshop* can be used for obtaining the resources.

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**. I

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.

Differences Between Versions 1.4 and 1.5

Undetected bugs were fixed in version 1.4 and 1.5 from version 1.3. First, for segments having **RET *number*** as the last instruction in the segment, the ***number*** was incorrect. Also, *HiLevel* previously contained a bug in its grammar. It rejected **.CODE** directives having more than two blank lines directly following them. This was undectected because *HiLevel* was only tested on files having **.CODE** directives without blank lines immediately following them. Finally, an adjustment was made to make relocatable offset and base address references somewhat more reliable, although they are in some cases incomplete, as described in the section on bugs.

In version 1.4, *Windows Disassembler* failed to handle programs in which there were empty relocation table(s), whereas in version 1.5 this has been fixed.

In version 1.5, **LROFFSET**s are now used instead of **OFFSET**s. **LROFFSET** forces the offset to be resolved by the loader at run time. If the offset is that of an exported procedure specified in the definition file, then using **OFFSET** will have the same effect. However, **LROFFSET** ensures relocateability regardless of whether the procedure is defined as exported.

The HiLevel Utility

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 **cCall**. The **cCall** 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 DxxxxH** instead of **xxxxH** when a number **xxxxH** follows **DS** in the parameter list of a **cCall** 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 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 fail to match, *HiLevel* generates a comment above the procedure that explains what is missing in the epilogue code. If the epilogue and prologue match, then the prologue/epilogue code is forced via the **FORCEFRAME** argument and/or **LOCAL** directive, plus by specifying any parameters. Otherwise, epilogue/prologue code expansion is prevented by not specifiying any parameters, locals, or prologue macro arguments.

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 50736 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 large procedures *HiLevel* will use up the local heap and "fly south" (hang). But for typical files, it should work. (Caution: HiLevel undoubtedly contains contains bugs, *therefore use it at your own risk*.)

As an example, the file **hellohil.asm** has been included. **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 **Procedure042A** 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 **Procedure03EB** and **Procedure03FA**. 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, **cCall**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 expands into the **LOCAL** directive. When it discovers, the **LOCAL** isn't there, it just continues assembling, but consequently distorts the expansion of the macro, to where either an error is generated or else garbage instructions are generated. The following listing demonstrates what happens when we change the **PUSH WORD PTR par4** and **CALL FAR PTR LocalFree** instructions in procedure **Procedure08A2** (in **hellohil.asm**) to a **cCall** macro call:

```
089D                    Procedure08A2     PROC  NEAR C <NOLOADDS, NOINCBP, FORCEFRAME,
NOCHECKSTACK>, par4:WORD
                                        cCall <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                        Procedure08A2     ENDP
```

Instead of **PUSH BP** and **MOV BP, SP**, it generates just **PUSH BP**, and the push specified in the **cCall** call doesn't get expanded either. Consequently, in *MASM 6.0*, *never call a macro or use 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 1.5

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 a 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                ; (Located in segment 0)
PUSH   WORD PTR D0AC0H
CALL   FAR PTR MakeProcInstance
```

This would be an error. The second **PUSH** should actually read, "**PUSH OFFSET Procedure00A4H**" The cause for this error hasn't been specifically determined.

License / Warranty Disclaimer

You are free to use, copy and distribute *Windows Disassembler* 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.

Copyright

Windows Disassembler and this documentation are copyrighted (c) 1992-1993 by Eric Grass.

Comments, critiques and suggestions regarding *Windows Disassembler* 1.5 are welcomed and can be forwarded to the following address.

Eric Grass
1612 Gettysburg Landing
St. Charles, MO 63303