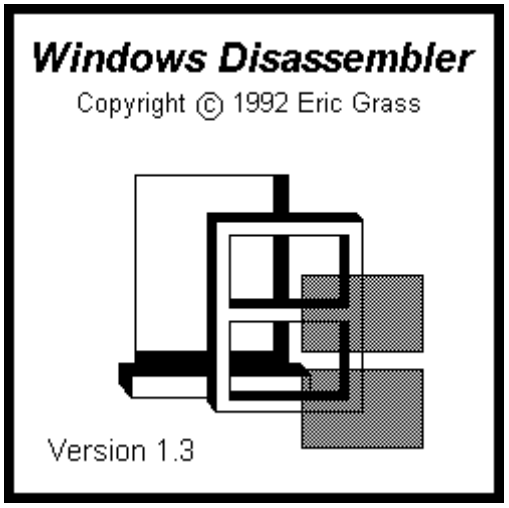
Windows Disassembler 1.3



User's Manual

Index

Introduction

Windows Disassembler disassembes Windows executables and dynamic link libraries. It allows you 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 you open a file when 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 you that the file does not meet a particular specification.

Displaying Assembly Language Source Code

Displaying code in the display window is presented as an alternative to generating a gigantic assembly language source code file, since some programs are bound to be quite large, and you may merely want to browse 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, you will typically see the numbers **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

Creating Assembly Language Source Code Files

After opening an executable, you can create an assembly language source code file for it using the **Save Text As** command. If the source code file name that you specify 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 you specify. 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 you specify 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 you specify **\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 you exit a segment which you have just modified, or if you try writing all of the segments to a file(s) at one time. However, if you use 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, you should set your editor's tab stop value 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)
```

Moreover, 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* 1.3 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.)

In addition, the user must figure out the entry point (used by the **END** directive) using a **.exe** file header utility as well. 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** and changing the directives into **EXTERNDEF**s. The **EXTERNDEF**s function 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 hello2.asm
link /ALIGN:4 hello1 hello2,hello2,, libw slibcew, hello.def;

which will generate **hello2.exe**. The alignment option prevents any further alignment, since the code has already been aligned by a C compiler.

Re-assembling medium, compact, and large model programs is more complex than the example given. The simplest way to disassemble and reassemble a medium/large-model program is to first save the segments in separate files (or modules). Then, in addition to the steps described above, do the following. 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 using an editor with a regular expression search function and replacing each occurance of "**^D**" with an "**EXTERNDEF D**" and "**DB 00[A-F,0-9][A-F,0-9]H**" with "**:BYTE**" (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, or any reverse resource compiler.

Potential Problems In Reassembling

One problem that usually will occur in reassembling is that of unknown labels due to references to labels that are located in a different procedure. The **::** operator must be used to make such labels global. Another potential 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 were thus interpreted as one string by *Windows Disassembler*. Simply seperate the strings. Finally, there is a potential problem related to modifying a program after it has been successfully disassembled and reassembled. The problem is that all address references are literal numbers. This is not a problem when you assemble in its original form, because the addresses don't change. But when you insert something new (or change something) in the middle of the code, the addresses of the instructions after get changed. What are needed before modifying the source code are symbolic addresses instead of literal numeric addresses. You must figure out which numeric operands are addresses by examining the context in which they occur, and then substitute the number with an **OFFSET *name***, **SEG *name***, **@Code**, or **@Data** directive, as appropriate. Finally, the error, "**A2006 : undefined symbol**" will occur when working with code having references to fixed functions outside of the given segment. This error is solved by supplying **EXTRN**s and **PUBLIC**s in the appropriate files.

Differences Between Versions 1.2 and 1.3

The recognizable differences in version 1.3 are the generation of **PROC** directives for near and far procedures as opposed to only far procedures, the representation of Windows API function names in their proper case, and the fonts option. All references (including offsets and segments) to exported functions and dynamic link library functions are now explicitly written, greatly reducing (and in many cases eliminating) the need for using an executable file header utilitiy. In addition, *Windows Disassembler* now handles references to fixed functions in addition to moveable ones. Also, the *HiLevel* utility, **hilevel.exe**, has been included with *Windows Disassembler* which partially converts *MASM* source code into C source code.

The HiLevel Utility

The *HiLevel* utility included with *Windows Disassembler* is a *Windows 3.1* utility which attempts to convert assembly language programs into C programs. 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** and **while** statements, **goto** statements, and basic assignment 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 attempts to construct function calls, although it cannot construct **long**/**double** arguments. If *HiLevel* cannot convert an instruction or group of instructions to C, it will simply write the instruction(s) in assembly code. If it encounters code that is not contained withing a procedure declaration, it writes out the code in assembly language within a comment using the delimiters **/*** and ***/**.

While *HiLevel* lacks features needed to generate pure C programs, it nonetheless can contribute something towards translating assembly language programs into C programs. 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 takes sometimes as much as a minute to process a source code file, and as long as you see 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 really huge procedures *HiLevel* will probably use up the local heap and "fly south" (hang). *HiLevel* was tested on the file **hello1.asm** plus five other larger syntatically correct files it and worked fine. A general rule-of-thumb is to assemble the source code first before using *HiLevel* on it, since there is no way that the program will ever compile if it fails to assemble to begin with.

The **while** loops constructed by *HiLevel* are not always quite correct. The problem is that the loop might include some of the instructions preceding the loop which are part of the test condition of the while loop, but *HiLevel* doesn't attempt to detect this.

With further development, it would be possible for this program to also construct **do-while**, **return**, **break**, **continue**, and **switch** statements, as well as to construct complex expressions (such as function calls passed as the arguments to other function calls, and so on) from *MASM* source code. However, such development will not be undertaken by the author at the present.

How HiLevel Works

HiLevel breaks the source code down into tokens and parses it, as a compiler would. Then, for each procedure, *HiLevel* constructs a directed flow graph data structure, where each node in the graph contains (points to) a block of code. A block of code is defined as a series of instructions in the source code where the first instruction in the block follows either a jump instruction (**JMP**, **JZ**, **JNZ**, **JLE**, ...) or a label, and the last instruction is either a jump instruction or else is followed by a labeled instruction. Moreover, all other instructions inside the block are non-jump, non-labeled instructions. That is, blocks are delimited by jump instructions and labels. (If *HiLevel* finds code outside of a procedure, it will write out the code inside a comment, one block at a time. Hence in this case you can see how *HiLevel* breaks up the code into blocks.) Each block is a linked list of instructions, and each instruction is stored in a tree, where the root of each tree is the instruction's mnemonic, the left child is its left operand, and the right child is its right operand. Since each block contains at most one jump instruction as the last instruction in the block, we say that the block jumps to another block provided that its last instruction is a jump instruction. Hence, to construct a directed flow graph for a procedure, *HiLevel* creates nodes for each block of code. Each block is then made to point to the block to which it jumps (if any) and also to the the next block after it in the code. *HiLevel* then test to see whether a given block is the start of an **if** statement by checking whether the other block to which it jumps is located somewhere after the original block in the source code and that the jump instruction involved is a conditional jump preceeded by a test instruction (**CMP** or **TEST**). If so, *HiLevel* indents and writes out all the code between the block and the block it jumps to, and then unindents and continues. Since there can be **if** statements within **if** statements, a recursive function is required to do this analysis. Detecting **while**, **do-while**, and **switch** statements is somewhat similar, although a bit more involved. As noted earlier, *HiLevel* handles only **if** and **while** statements.

Constructing function calls involves checking for a series of **PUSH**es followed by a **CALL**. However, this can get complicated if there are instructions in between the **PUSH**es of a function call for evaluating an expression before passing it as an argument. The solution would be to maintain (a) variable(s) which keep track of the logical values of the **AX** and **DX** registers (whether they be a return value from a function or else an arithmetic expression) and check whether the

instructions between two **PUSH**es can be reduced to a single expression and whether the second **PUSH** following the instructions pushes that expression/value. Since functions can have functions has arguments, which in turn can also have functions as arguments, etc., a recursive function would be required here as well. *HiLevel*, however, only goes so far as detecting functions with **PUSH**es that immediately follow one another

Bugs

Known Bugs In Version 1.3

The screen will need refreshing occassionally after scrolling upwards, mainly within data segments, but sometimes in code segments if you edit the bytes. This bug is minor and will not affect file generation.

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

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

```
PUSH   OFFSET ABOUTDLG
PUSH   OFFSET ABOUTDLG
PUSH   WORD PTR D0AC0H
CALL   FAR PTR MakeProcInstance
```

Obviously this is an error. The first **PUSH** should actually read, "**PUSH SEG ABOUTDLG**." This cause of this bug is unknown.

License / Warranty Disclaimer

You are free to use, copy and distribute *Windows Disassembler* providing 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. In particular, please note the copyright terms of the program you process.

Copyright

Comments, critiques and suggestions regarding *Windows Disassembler* 1.3 are welcomed and can be forwarded to the following address.

Eric Grass
1612 Gettysburg Landing
St. Charles, MO 63303