Windows CodeBack™

Version 1.05

User's Manual

Table of Contents

0. NOTES

- All company and product names mentioned through this document are registered trademarks or trademarks of their respective holders.
- I would like to say thanks to all people who helped me during the development of WCB, especially to Endre Csató for impressive testing and for good advices, to Árpád Csoma for suggesting the **-S** option, to Károly Kecskeméti for giving the CodeBack name and for his hard work in spreading WCB over the world, and to László Kôvári for putting the previous versions onto all BBSs that he could reach.

1. OVERVIEW

What is WCB

Windows CodeBack (WCB) is a disassembler designed exclusively for Microsoft Windows applications. It can disassemble New Executable format files (such as .EXEs, .DLLs, .DRVs, etc.), Linear Executables (.386s and .VXDs) and can extract LE files bounded into W3 files (such as WIN386.EXE). The current version does not include support for conventional DOS .EXE files, OS/2 2.x LX, and Win32 PE files, but you can disassemble OS/2 NE files too (Notice, that if you want to disassemble OS/2 NE files, you need an export list for the DOSCALLS module, so please generate one before disassembling. More on generating export lists later.)

After you start it, WCB gathers and shows information about the program's entry point, main procedure, device descriptor block. Shows by name all Windows API functions that an application call and all the virtual device driver calls that a VxD makes. Labels the exported functions in a program and the control procedures and services in a VxD. Identifies by name and labels the WinMain and LibMain functions. Show General Protection fault handlers found in a __GP block, and apploader functions in a self-loading application. Uses CodeView symbols, such as those shipped with the debugging version of Windows.

Hardware and Software Requirements

In order to run WCB you will need at least a 386 based machine with DOS 3.0 or above installed on the system, an XMS memory manager such as HIMEM.SYS (the XMS version it provides must be 2.0 or higher.) And it is not bad if you have a plenty of free hard disk space (from giant applications such as WinWord WCB may create a 20-30 MB list file!). WCB runs correctly in the Windows NT command prompt window.

The total amount of memory WCB need depends on the largest segment's and overall size of the program to be disassembled. For example in the case of WinWord about 250k conventional and 200k XMS needed; in the case of the debug version WIN386.386 (embedded into WIN386.EXE) 260k conventional and about 150k XMS will be sufficient (if you use the code analyzer, some more memory needed).

Installation

There is a very simple installation tool provided in the registered WCB package. This is the INSTALL.BAT file that you can find on the WCB diskette. So if you want to install WCB, make the drive containing the WCB distribution diskette the active drive (by typing A: for example), and start the install program by typing INSTALL. If you don't want to install WCB into the default C:\WCB directory, you may specify the desired path on the INSTALL command line, for example by typing INSTALL D:\WCB. After a little disk crackling you will get the Windows CodeBack installed on your system, and it's ready to play with.

A Quick Start with WCB

You can start WCB on the DOS command line by typing:

```
WCB [ options ] module [ listfile ]
```

or

`WCB @respfile` if you want to use a response file.

WCB searches for the given module with the following extensions if you did not give one: .EXE, .DLL, .DRV, .VXD, .386. When you generate a disassembled list from a module, the name of the listfile (if you didn't specify one) is the name of the module file with the extension .LST. If you specify the listfile name but no extension, then the extension will be .LST. When you are generating an export list file, all is similar to the previous case, but the default extension will be .EXL.

Options can be preceded either with '-' or '/', but notice, that the option letters are case sensitive. You can place options anywhere on the command line, for example:

```
wcb -c test
wcb -s2 test -c
wcb test -c test.lst -r
wcb test -c -r
```

are all valid.

If you don't like bothering with the command line, you can use a response file. This file contains the WCB command line, each piece separated by at least one space or a new line character as you can see in the following example (contents of EXAMPLES\VMD\VMD.WCB). The default extension for the response file is .WCB.

```
\windev\vmd\vmd.386
-c
-y
-r
```

2. File Formats

WCB uses several types of text files. These are the following:

Export Lists (.EXL)
Output List (.LST)
VxD Service List (.VSL)
Region Definition File (.RGD)

The region definition file format is documented in the *'4. Regions'* section, and the others here.

Export List File Format (.EXL)

The format of the export list file is the following:

<module name>
[<ordinal> <function name>] ...

where

<module name>	is the name of the module the .EXL file belongs to.
<ordinal>	is the ordinal number of the exported function.
<function name>	is the name of the exported function.

You can generate an export list from a module with the **-x** switch. And now a few words about the customization of .EXL files. WCB searches for .EXL files at first in the current directory then in the XXL directory (specified with the WCBXXL environment variable or with the **-E** option), so you can make your own version of an export list file, place it in your working directory, and start WCB: it will read your customized file rather than the one can be found in the XXL directory.

The following is an example that illustrates how the pervious things look like in reality (the contents of XXL\LZEXPAND.EXL provided with WCB):

```
LZEXPAND
     1 LZCOPY
     2 LZOPENFILE
     3 LZINIT
     4 LZSEEK
     5 LZREAD
     6 LZCLOSE
     7 LZSTART
     8 COPYLZFILE
     9 LZDONE
    10 GETEXPANDEDNAME
    11 WEP
    12 ___EXPORTEDSTUB
```

Output List File Format (.LST)

The .LST file is the default listing file generated from a module (unless you used **-x** switch). There are three main sections and several subsections in an .LST file, as you can see in the following examples.

Example 1 (from EXAMPLES\TEST\TEST.LST)

```
Filename:               TEST.EXE     ⇐ This is the header section.
                                        Use -nh to suppress.
Type:                   Segmented executable
Module description:     Windows Startup Test
Module name:            TEST

Imported modules:                    ⇐ The imported modules section.
  1: KERNEL                             Use -ni to suppress.
  2: USER

Exported names by location:          ⇐ The exported names section.
  1:0130     1 WNDPROC                  Use -ne to suppress.

Program entry point:    1:0000       ⇐ Entry point info block.
WinMain:                1:0050          Use -np to suppress.

-- Segment: 1 -- Type: 16 bit ------------------------
:
:
```

Example 2 (from EXAMPLES\VMD\VMD.LST)

```
Filename:               VMD.386      ⇐ This is the header.
Type:                   386 enhanced mode virtual...
                                          ...device driver
Module description:     Win386 VMD Device  (Version 3.0)
Module name:            VMD

Imported modules:                    ⇐ The imported modules.

Exported names by location:          ⇐ The exported names.
```

```
00000AA4     1 VMD_DDB

Segment names:                       ⇐ This is the segment names
  1: _LGROUP                            section. Use -nn to suppress.
  2: _IGROUP

Symbols by location:                 ⇐ This is the symbol list.
00000000  Int33_Create_VM               Use -ny to suppress.
:
00000200  I33_Soft_Init_INI

Program entry point:   3:0000        ⇐ This is the DDB info block.
Device descriptor:   00000AA4           Use -np to suppress.
Device number:       000C
Device version:      03.00 (3.0)
Init order:          34000000
Control procedure:   0000064C
V86 API procedure:   00000826
PM API procedure:    00000826
V86 API entry CS:IP: N/A
PM API entry CS:IP:  N/A
Service table:       00000A98
Number of services:  00000003 (3)

-- Segment: 1 -- Type: 32 bit -- Name: _LGROUP -------
:
:
```

The first is the header section, which contains information about the file name and type, module name and description. The second is the information section that informs you about the imported modules, exported names, program entry point, WinMain/LibMain and in the case of VxDs the device descriptor block.

The third is the core of an .LST file. This is the disassembled (and if you specify the **-d** switch the dumped) list of segments - or objects if you are disassembling a linear executable. (For simplicity I will refer to objects as segments in the following.) If you use the **-S** switch, only the first two sections will appear in the .LST file; the segment lists will be placed into .xxx files, where xxx is the zero padded number of the segment. In the list each segment's header contains some more information: the number of the segment, type of the segment (whether it is 16 or 32 bit), and segment name (if present in the .SYM file).

After the header begins the disassembled list of the segment. There is some extra information in this list along with pure assembly language instructions. This extra info appears in a form of comments. There is a comment in the list everywhere an exported function, control procedure or service routine begins, but this is self-explanatory. There is an other comment type that I will discuss shortly in the following. If you encounter a comment **Reloc => <relocname>** in the disassembled list, it means that the segment's relocation table contains a reference to the following address with the name **<relocname>**. If the **<relocname>** is a function name it means that there is code at that address, so the code analyzer made some trash; but if the name is a variable name (like KERNEL.__WINFLAGS), then all is in order.

Another strange thing you might encounter in the listing file is a ? in the place of the instruction's mnemonic. This little ? appears when the disassembler finds an invalid opcode (that is not specified in the i486 Programmer's Reference). Almost always it means that there is data rather than code, but there are exceptions. One of them is the 0fh 0ffh sequence, which the Windows 3.1 kernel uses to switch from one protected mode ring to another.

WCB show General Protection fault handlers in the output list too. For more information on this topic, please see the *General Protection Fault Handlers* subsection in the *'5. The Code Analyzer'* section.

VxD Service List File Format (.VSL)

Files of this type are used to replace the ugly numbers in a VxD call with more meaningful function names. WCB can't generate this type of file, so you must provide it manually or use the default file included in the WCB package. Every time you start WCB it reads the standard .VSL file called DEFAULT.VSL, which you can find in the XXL directory. You may replace this default file: make the necessary changes and use the **-l** switch to specify the new location of the file. The format of a .VSL file is very simple:

[<service number> <service name>] ...

where

<service number>	is the 8 digit hexadecimal number of the service.
<service name>	is the name of the service.

The following is a snippet from the XXL\DEFAULT.VSL file, that illustrates the VxD service list file format:

```
:
00010079 _Free_LDT_Selector
0001007A _BuildDescriptorDWORDs
0001007B _GetDescriptor
0001007C _SetDescriptor
0001007D _MMGR_Toggle_HMA
0001007E Get_Fault_Hook_Addrs
0001007F Hook_V86_Fault
00010080 Hook_PM_Fault
00010081 Hook_VMM_Fault
00010082 Begin_Nest_V86_Exec
00010083 Begin_Nest_Exec
00010084 Exec_Int
00010085 Resume_Exec
00010086 End_Nest_Exec
:
```

3. WCB SWITCHES

The following is the alphabetical list of the currently available WCB options and switches. Note, that some of them (**-oyl**, **-oyn**, **-t**, **-v**, **-y**, **-l**) are available in registered version only. For more about registering WCB see 'REGISTER.DOC' file.

-c If you use this switch, a code analyzer will be used to separate code from data rather than to decide on a code segment/data segment basis. See *'5. The Code Analyzer'* and *'4. Regions'* for more on this topic.

-d Use this switch if you want to dump all segments after disassembling.

-D With this switch you can force WCB to treat data segments as if they were code during the code analyzing process; so then WCB will accept references for code pieces in data segments too. Note that this switch takes effect only if you use the code analyzer (switch **-c**).

-en With this option you can specify the ending segment of the disassembling process. If you use both **-e** and **-s**, and the ending segment number is less than starting, WCB will not process any segments.

-Exxx With this option you can tell WCB that where is the XXL directory. See also *'The WCBXXL variable'* subsection in this section.

-Ixxx By default a file named DEFAULT.VSL is used to retrieve the names of VxD services. If you want to use your customized version of this file, use this option to tell WCB where to look for it.

-L This switch tells WCB to display a license agreement screen.

-ne This switch suppresses the list of the module's exported names. See *'2. File Formats'* for more about listing files.

-nh This switch suppresses the header information in the listing file. See *'2. File Formats'* for more about listing files.

-ni This switch suppresses the list of imported modules in the listing file. See *'2. File Formats'* for more about listing files.

-nm By default in a listfile a reference to a Windows API function appears in **MODULE.FUNCTION** format (like KERNEL.MAKEPROCINSTANCE). If you use this switch, the module names and the dots will be stripped from these references (so from KERNEL.MAKEPROCINSTANCE will be simply MAKEPROCINSTANCE). See *'2. File Formats'* for more information on listing files.

-nn If you are using a symbol file, and the .SYM file contains symbols for segment names, this switch causes that these names won't appear in the information section of the listing file. See *'2. File Formats'* for more about listing files.

-np If you use this switch with Segmented (or New) Executables, addresses of the entry point and WinMain or LibMain will not appear in the listing file. With 386 enhanced mode virtual drivers, the effect is similar to the previous case: there will be no Device Descriptor info block in the listing. See *'2. File Formats'* for more information about listing files.

-nw This switch tells WCB, that don't search for the WinMain/LibMain functions. This is a useful switch if you have a non-standard executable or library, that have no conventional startup code and main procedure pair.

-ny This switch suppresses the list of symbols that came from the .SYM file. See *'2. File Formats'* for more about listing files.

-ol If you use this switch, the names in the export list will be sorted by their location.

-on If you use this switch, the names in the export list will be sorted by export name. This is the default case when you disassemble a module.

-oo If you use this switch, the names in the export list will be sorted by ordinal number. This is the default case when you generate export list from a module with the **-x** switch.

-oyl If you use this switch, the names in the symbol list will be sorted by location. Note, that this option (and the following one) is available in registered version only.

-oyn If you use this switch, the names in the symbol list will be sorted by name. This is the default state of the **-oy** switch, and is included only for completeness.

-pn This option specifies the number of disassembling passes. The minimum is 2, the maximum is 9, and the default value is 3. See *'5. The Code Analyzer'* for more on passes.

-rxxx Specifies the name of the optional region definition file. If you do not specify the file name (that is only **-r** used), the module file name will be used with the extension .RGD. See *'4. Regions'* for more information on regions and region definition files.

-sn With this option you can specify the starting segment of the disassembling process. If you use both **-e** and **-s**, and the ending segment number is less than starting, WCB will not process any segments.

-S By default the disassembled lists of all segments will be placed into one listing file. With this switch you can tell WCB to place each segment list in a separated file. In this case the .LST file contains only the information block of the listing file, and segment listings are placed into .xxx files, where xxx is the number of the current segment. If you disassemble large files with many segments (like WinWord or CorelDraw) this switch is highly recommended. Note that you can use this switch with programs that has maximum 999 segments, but the maximum number of segments is 254 in the case of New Executables and 3 if you have a virtual device driver, so it's only a theoretical limit. See *'2. File Formats'* for more about listing files.

-t If you use this switch, WCB lists all the VxDs that can be found in a W3 file (like WIN386.EXE) rather than disassembling.

-vxxx If you want to disassemble a VxD that can be found in a W3 file (like WIN386.EXE), you must specify the VxD name with this option.

-yxxx This option can be used to specify the optional symbol file name. If you do not specify file name (that is you use **-y**) then the module file name will be used with the extension .SYM. The symbol file contains additional symbolic and segment names that eases the interpreting of disassembled code. .SYM files generated with Microsoft's MAPSYM or Borland's TMAPSYM can be used. If the given file is not a valid .SYM file, the program may crash, because a .SYM file doesn't contain a signature, so WCB can't decide that it is valid or not. Note that this option is available only in the registered version.

-x This switch tells WCB to produce export list (.EXL) rather than a listing (.LST) file. See *'2. File Formats'* for more about listing and export list files.

The WCBXXL variable

WCB searches for export list and VxD service list files in a directory called the XXL directory. With the WCBXXL environment variable you can tell WCB the name of the XXL directory.

4. Regions

A region is a contiguous area of code or data. The size of a region can range from a single byte to a full segment. Separating the disassembled list into regions may significantly ease the interpreting process. There are three ways to tell WCB where are code, and where data regions:

- By default WCB uses the segment attribute to decide that the actual segment is a code or data segment. If code, the full segment will be code region; if data, the full segment will be data region (in the case of VxDs all the segments are code).

- You can use the code analyzer and let WCB to discover code and data regions. Unfortunately the code analyzer is not perfect and sometimes may mark code regions as data (in the case of call/jmp with indirect register and memory addressing modes). If you encounter this problem, you must make a region definition file. See also option **-c**.

- If you want to mark an area to code or data region manually, you can do this by making a region definition file. You can tell WCB the file name by using the **-r** option. The format of the region definition file is the following: each line in the file specifies a new region. The format of the lines is:

  <type> <segment> <beginning offset> [<ending offset>]

  where

<type>	is the type of the region. 'C' means code region, 'D' data and 'A' autodetect. (Autodetect is a code region with automatic length detection.)
<segment>	is the segment of the region.
<beginning offset>	is the offset within the segment, where the region starts.
<ending offset>	is the offset within the segment, where the region ends. This is not needed for type 'A' regions. In this case (type 'A') the code analyzer will be used to discover the length of the region.

Note that type 'A' can be used only with option **-c**.

Here comes an example. Suppose that you want to disassemble a library called WHATIS.DLL. This library contains indirect jumps or calls, so the code analyzer makes a lot of trash. You analyzed the correctly disassembled pieces of the code, and found that a routine begins on 1.13f, 1.3ee, 1.2e2a; code must be disassembled from 2.1003 through 2.8873 and data must be disassembled from 2.8874 to 2.88fd. So it's time to make the region definition file like this:

The contents of WHATIS.RGD:

```
a 1 13f
a 1 3ee
a 1 2e2a
c 2 1003 8873
d 2 8874 88fd
```

After the successful creation of this file you can invoke WCB by typing:

```
wcb -c -r whatis
```

and you will get the correctly disassembled list of WHATIS.DLL.

5. THE CODE ANALYZER

The code analyzer is used to separate code from data in a segment. In the case of New Executables this is rarely needed, because a bit in the segment attribute specifies the type of the segment. In some cases however a code segment may contain data (and vice versa). This is the case with Linear Executables: the image of an LE file does not contain any type of information where are code and where data regions in the segment.

The code analyzer takes the entry point of the program, addresses of all the exported functions (in the case of NE files) or addresses of control procedures and all the services that a VxD provides, and inserts these addresses into a priority queue. The analyzer then gets an address from this queue and begins disassembling. If it finds a **ret**, **retf** or **jmp** instruction then marks the previously walked area as code; if it finds a **j**xx, **loop**, or **call** instruction then stores this instruction's argument into the priority queue: if this address is in the current segment then inserts it with high priority, otherwise with low. WCB displays the number of elements in the queue during the analyzing process: don't get frightened if this number sometimes goes up, this means that there were many branches in the last region. The code analyzer currently does not process register/memory indirect jumps/calls. In the case of a VxD, some function calls are treated as if they were **call** instructions (ESI contains the address). These function calls are:

```
0001000b  Allocate_V86_Call_Back
```

0001000c	Allocate_PM_Call_Back
0001000d	Call_When_WM_Returns
00010010	Call_Global_Event
00010011	Call_VM_Event
00010014	Call_Priority_VM_Event
00010017	Set_NMI_Handler_Addr
00010018	Hook_NMI_Event
00010019	Call_When_VM_Ints_Enabled
00010024	Call_When_Not_Critical
0001002a	Call_When_Task_Switched
0001003a	Call_When_Idle
0001003d	Set_VM_Time_Out
00010041	Hook_V86_Int_Chain
00010071	Hook_V86_Page
0001007f	Hook_V86_Fault
00010080	Hook_PM_Fault
00010081	Hook_VMM_Fault
00010090	Hook_Device_Service
00010091	Hook_Device_V86_API
00010092	Hook_Device_PM_API
00010095	Install_Mult_IO_Handlers (table address in EDI)
00010096	Install_IO_Handler
000100bf	Fatal_Memory_Error (treated as a **jmp**)
0003000f	VPICD_Call_When_HW_Int
00170004	SHELL_Message

In some cases the analyzer might do some 'trashing': it might mark code areas as data. In these cases the user may mark explicitly these areas as code using region definition files (this feature is discussed in the *'4. Regions'* section).

Don't forget to specify the **-c** switch on the WCB command line if you want to use the code analyzer.

More on Passes

Here I must discuss another code analyzing facility of WCB: the multi-pass disassembler. The number of passes is what you can specify with the **-p**n option. The minimum is 2, the maximum is 9, and the default value is 3.

During the first pass, WCB puts the operands of all the **jmp**/**call**/**j**xx... instructions and addresses of all exported functions (and similarly, addresses of control procedures and all services in the case of a VxD) into a table. This table will be used during the later passes as a synchronizing table: the disassembler will always begin a new instruction at an address that occurs in this table. Of course the sync table will be updated during each pass (except the last, when WCB writes out the list file). Here follows an example to clearly understand the previous things:

Suppose, that you have the following code (from Windows 3.1 KRNL386.EXE): During the first pass it looks like:

```
1.00CC  8ED8            mov     ds, ax
1.00CE  C3              ret
1.00CF  00558B          add     byte ptr [di-75], dl

                        ;
1.00D0                  ;  ALLOCCSTODSALIAS
                        ;

1.00D2  EC              in      al, dx
1.00D3  68E100          push    00E1
```

This is obviously bad. But after the sync mechanism was used the code looks much pretty:

```
1.00CC  8ED8            mov     ds, ax
1.00CE  C3              ret
1.00CF  00558B          add     byte ptr [di-75], dl

                        ;
1.00D0                  ;  ALLOCCSTODSALIAS
                        ;

1.00D0 >55              push    bp
1.00D1  8BEC            mov     bp, sp
1.00D3  68E100          push    00E1
```

You may see, that the more passes you use, the more accurate code you will get (but I think 5 is fairly enough).

Note that you can see a '>' after each address where the sync table was used.

What about WinMain and LibMain

When you start a Windows program or Windows loads a library, Windows does not directly transfer control to the program's or library's main function called WinMain in the case of applications and LibMain in the case of dynamic link libraries. There is a small piece of code, called the startup code, where the execution begins. Then this code will call WinMain or LibMain. But which **call** instruction is that calls this main function? And which is this function anyway? I had examined Borland's and Microsoft's startup code along with many applications' and found that:

- WinMain is the first function (far or near) that is called with five words pushed onto the stack after a call to USER.INITAPP.
- LibMain is the first **far** function that the startup code calls after a call to KERNEL.LOCALINIT.

So WCB determines the presence and location of WinMain and LibMain these ways.

If the program you want to disassemble doesn't contain this conventional startup code/main procedure pair, then you may use the **-nw** switch to tell WCB not to search for WinMain/LibMain, so WCB will not mark a function as a main function incorrectly.

General Protection Fault Handlers

In the Windows 3.1 KERNEL, USER and GDI modules you can find blocks of code that are protected from GP faulting. This is a part of the 3.1's parameter validation mechanism: if you pass a NULL pointer, invalid selector, etc. to a Windows API routine, but it won't cause a GP fault, so the system will remain in a stable state. WCB shows the General Protection fault handlers (which you can find in a table under the exported name __GP) in a format you can see in the following code snippet:

```
3.0184  0BD9           or      bx, cx
3.0186  7411           je      0199

                       ; GP block - handler at 3:019A

3.0188 >0F02D9       ⌈ lar     bx, cx
3.018B  7510         | jne     019D
```

```
3.018D  F6C708       │ test    bh, 08
3.0190  740B         │ je      019D
3.0192  8EC1         │ mov     es, cx
3.0194  8BD8         │ mov     bx, ax
3.0196  268A07       └ mov     al, byte ptr es:[bx]

3.0199 >C3              ret

                       ; GP handler to block...
                                   ... 3:0188 - 3:0199

3.019A  83C404      > add     sp, 0004

3.019D >BB0870        mov     bx, 7008
3.01A0  EB00          jmp     01A2

3.01A2 >5A            pop     dx
3.01A3  0E            push    cs
3.01A4  52            push    dx
3.01A5  EA9B941801    jmp     K327
```

6. ERROR MESSAGES

All available XMS handles are allocated.

You are using XMS memory extensively, so increase the number of XMS handles in your CONFIG.SYS file with the **/numhandles=**n switch. In most cases the default 32 is a pretty good number.

Autodetect type regdefs can't be used without the code analyzer.

Your region definition file contains an autodetect type regdef but the code analyzer is not enabled. Use the **-c** option or remove autodetect type regdefs from the .RGD file.

Can't open export list *filename*

This message appears if WCB didn't find an export list (.EXL) file for a module that is referenced in the executable or library you want to disassemble. To avoid this problem, please generate an export list from the referenced module with the **-x** switch.

Can't open module *filename*
Can't open region definition file *filename*
Can't open response file *filename*
Can't open VxD service list file *filename*

These errors might occur if an input file was not found. Please check that the given filenames are correct.

Can't open listfile *filename*
Can't open segment list file *filename*

These errors might occur if an output file can't be opened because of a file system or disk error. If you encounter any of the above error messages, please check the given file name, that no directory with the same name exists, the output file does not exist with read only attribute, etc.

Disk full.

You have no more free space on your disk. Please delete or compress some files, and run WCB again.

filename* does not contain symbols for module *modulename

You specified a symbol file with the **-y** switch, but a wrong one; that file doesn't contain valid symbols for the current module.

DOS Version 3.0 or later required.

WCB requires MS-DOS version 3.0 or later. But who are not using DOS 5 in these days!?

Error in region definition file.

An error occurred in the region definition file: an invalid segment number specified, invalid regdef type specified or the given address is out of segment boundaries. Please check the specified regdef file.

HIMEM.SYS or other XMS driver required.

WCB requires XMS memory in order to run, so please insert an XMS driver into your CONFIG.SYS file.

Invalid switch *switch*.

You specified an invalid switch on the WCB command line. Please check your command again.

***filename* is not a valid self-loading executable.**

The specified executable's header flag tells WCB that it is self-loading, but the first segment doesn't contain the 'A0' signature.

***filename* is not a valid Windows module.**

You tried to disassemble a file that is not a Windows module, that is not a NE, LE, W3 or PE file.

Linear Executables supported in registered version only.

This message appears if you have a non-registered version of WCB and tried to disassemble an LE, or W3 file. For more about registering WCB see the REGISTER.DOC file.

Module *filename* is not a W3 file.

You used the **-v** or **-t** option with a file that is not a W3 file. Please check the filename.

No export list for module *modulename*

No export list exists for an imported module. Please use WCB with the **-x** switch to generate one.

No VxD name specified; use the -v switch.

You specified a file name that is a W3 file, but WCB can only disassemble the VxDs embedded into the W3 file, so use the **-v** switch to select one of them.

Not enough conventional memory.

You are out of conventional memory. Please remove some TSRs from AUTO-EXEC.BAT or use a memory manager, for example EMM386 or QEMM.

Not enough XMS memory.

You are out of XMS memory. Remove some XMS eating programs (like disk caches), or buy more memory.

One or more of the used options available in registered version only.

You specified an option that is available in registered version only. For more about registering WCB see REGISTER.DOC file.

Please use Win32 CodeBack to disassemble PE files.

If you would like to disassemble PE files, please use Win32 CodeBack, which is a separate product, and is not included in the WCB package.

Stack overflow occurred, please inform the author.

This is a fatal internal error, so please report it to me.

Symbol file *filename* not found.

The specified symbol file not found, so check the given filename.

This program needs a 80386 or higher processor.

WCB was compiled to take advantage of the 386's 32 bit registers and addressing modes, so you will need a 386 based machine to run WCB.

VxDname* VxD not found in *filename

You specified a VxD name with the **-v** switch, but no VxD with this name exists in the given file.

XMS error *code*, please inform the author.

This is a fatal internal error, so please report it to me.

XMS Version 2.0 or later required.

You are using and old version of HIMEM.SYS. Please upgrade to a newer one. Those shipped with MS-DOS 5 or Windows 3.1 are suitable ones.

7. EXAMPLE PROJECTS

There are three example projects included in the registered WCB package. The first of them is a simple Windows program, the second is the disassembled list of Windows 3.1 Task Manager and the third is the disassemblation of the debug version Windows 3.0 Virtual Mouse Driver. You can find these examples in the 'WCB\EXAMPLES' directory.

And now some words about these examples. The first and simplest example (which you can find in the WCB\EXAMPLES\TEST directory) is a simple Windows application that only displays a window on the screen, processes WM_PAINT messages for the window, but does nothing useful. All the source files for the TEST application included. Along with the necessary .C, .H, .DEF files you can find a Borland C++ project file used to build the app, and a minimal startup code module. This was the application that I disassembled most often during the development of WCB.

The second example is the disassemblation of Windows 3.1's Task Manager, for which the necessary files are in the WCB\EXAMPLES\TASKMAN directory, including the .LST and .WCB files.

The third example is more interesting than the previous ones (at least I think). This is the disassembled list of the debug version Windows 3.0 Virtual Mouse Driver included in Windows 3.0 DDK. This example demonstrates the use of .SYM and .RGD files too. With examining this example you can learn about how to disassemble VxDs, and how a VxD works; however I think you must deeply examine the DDK documentation. Oh, I almost forgot that you can find this example in the WCB\EXAMPLES\VMD directory.

8. Recommended Reading

Along with the SDK and DDK documentation, there's a lot of good books in the bookstores on the subject of understanding the inner workings of the Windows operating environment. In the end of this little manual I will mention some of them.

- **Undocumented Windows** (by Andrew Schulman, David Maxey and Matt Pietrek; Addison-Wesley ISBN 0-201-60834-0): This is a fundamental book for everyone who likes to know what's going on under the hood of Windows. I like especially the 3rd Chapter: Disassembling Windows, which inspired me to write my own disassembler.

- **Windows Internals** (by Matt Pietrek; Addison-Wesley ISBN 0-201-62217-3): With providing a pseudocode for the main Windows API routines, this book is a goldmine for every programmer. Along with the pseudocode routines, the book thoroughly explains the booting and shutting down of Windows.

- **Writing Windows Device Drivers** (by Daniel A. Norton; Addison-Wesley ISBN 0-201-57795-X): If you haven't got the DDK, this book gives you the overview of the Windows device driver layer, and describes a lot of VxD services.

And lastly... have a lot of fun with Windows CodeBack...