VBMDis - DoDi's VB Make Discompiler

VBMDis is a direct descendent of **VBDis**, designed to help you optimising your programs. It allows the **comparison** of both your sources and the compiled program. Examining the **tokens shown** and the definitions of the variables found in the EXE, you can find how your program may become smaller and faster. In revision 3.38 support was extended from VB 3.0 to **VB 2.0**, but some methods may be handled incorrect for VB 2.0.

Installation of **VBMDis** is easy, simply copy the whole directory to your hard disk or expand the archive in a new directory. No setup program overwrites your VBXs and DLLs, and your INI files are not touched. To remove VBMDis, delete the directory, and it disappears without leaving a trace.

Usage

Compile your VB program and open the **make** file in **VBMDis**. Then the Discompiler starts (just like with **VBDis**), after this you'll get two windows with the modules in your sources and in the executable. The windows contain combo boxes to select a module and a subroutine, and list boxes to show the source text and the tokens for any line. You may adjust the height of the listboxes just as with the interpreter, the mouse pointer changes its shape over the regions designed for this.

The sources must be saved in binary format, to allow the display of the tokens.

If you use to save your projects **as text** (as is required with **VBPro**), you're recommended to create a **copy** of your project, using **VBPro**'s option Project|Create|Copy, and use VB to save all modules in **binary format**.

The Windows of VBMDis

The **main window** is a MDI window, after opening a project you'll find there two windows with your source modules and the executable. In the menu, you can open a third window with any text file. While analysing the executable, the window known from **VBDis** appears, and you can open two more windows with the descriptions of the global and local variables in the executable.

The menu option Window|Wide toggles the arrangement of the make and exe windows in the MDI window. You can see long lines better with the windows below each other (with the full width of the main window), and better find matching lines if they are arranged side by side.

You **select** modules and subroutines separately for both windows, in the exe window you'll find an additional module with all global declarations, that cannot be assigned to a specific module. The lower part of each window shows the **text** of the subroutines and declarations. If you select a line from it, the **tokens** forming the line are shown in the list box above the text. Every token line contains the name of the token and its arguments (in hex). In the exe window, no tokens are available for the declarations section and the declarations of subroutines and local variables and constants (you find such informations in the separate definition windows). Instead, for the same line you may find more tokens in the exe window than in the source window. These tokens are added by the compiler, sometimes they are only named **exe**, but you may also find added type conversions (**C<type>**).

The **exe tokens** have an added description of the **runtime stack** and the changes occurring when the token is executed. In addition to the type chars of VB (**%&!#@$**), the token arguments may be of the types **A**rray, **O**bject, **T**ype or **v**ariant. **Pointers** are prefixed with a caret (**^**), but arrays, types, objects and strings are always handled by implicit pointers (not flagged). If a string is passed **ByVal** to a subroutine, a pointer to a copy of the string is passed as argument. That's why the Discompiler cannot decide whether a string is passed ByVal or ByRef, if the subroutine is never called.

The **stack arguments** and **operations** of a token are listed in the second column, in a function-like notation. The 'name' of the function is the data type pushed to the stack, the 'arguments' are the arguments' types popped off the stack.

Example

v($,%) means a string and an integer are popped off the stack, and a variant is pushed instead.

More information is given if a token uses a **variable** or calls a **function**. Both such arguments are shown with its scope and offset, the characters meaning **g**lobal, **m**odule, **p**arameter or **l**ocal variables. For parameters, **pv** denotes a By**V**al **p**arameter. The hex number following the type char is the offset of the description of the variable or function in the modules declarations. With a double click on such a line you can **view** that description in the separate module declaration window.

Example
l1234%() means that a local integer variable at offset &H1234 is accessed.

Access to a **module** variable or constant goes straight to the offset shown. For **global** variables or constants however, there stands the offset into the global declarations. For **parameter or local** variables, there stands the offset into the execution stack, relative to the subroutines base offset. For locals and **ByVal** parameters there stands the value, for **ByRef** parameters again a pointer to the value is stored there. So I think, using a module variable (Dim or Static) should give the fastest access, while locals and ByVal parameters need one more step, and access to ByRef parameters is the slowest, with a double indirection. An exception is found with local and parameter **strings**, marked with positive odd offsets, presumable in another segment. **Variants** include a normally invisible string, whose number is appended to the variants description.

Unused local variables and parameters can be recognised by an offset 0000.

In the **stack display** you may find superfluous type conversions, as shown with **Mid** and **Mid$**:

The assignment
```
s$ = Mid$(x$, y%)
```
directly pushes a string, whereas
```
s$ = Mid(x$, y%)
```
first pushes a string, because only **Mid$()** is implemented in the runtime system. That string is then converted to a variant (shown as **v($)**), what were the result of Mid(). Before the assignment to the string variable can be made, another conversion (shown as **$(v)**) is needed.

Unnecessary conversions may occur with constants and Single variables, too. You may find the best coding by putting different versions of a statement in sequential lines of your source and examining the tokens created by the compiler. Variables of type Single are useful only in arrays, to save space. Such variables are always pushed as Doubles, you may encounter additional type conversions (shown as **#(!), !(#)** or **C<type>**).

Buttons in the Code Windows
The **sequence** of the modules may be different in the make file and the executable file. In the make window, there are two buttons named match. The **upper** one beneath the module combo box copies the name of the module and all subroutines to the exe window, the **lower** button copies the name of the selected subroutine, if required. Normally the order of the modules is the same in both windows, with the exception of the start form that occurs in the first place in the make window. The names of all **forms** are retained in the executable, so only other modules may produce problems. You can put a constant string with the module name into every module, that will be displayed in the declaration section of the exe modules.

The Rename button in the exe window assigns the current module the name you entered in the text field of the module combo box. The Variables button opens and refreshes the window with the description of the module's variables, the global descriptions are accessible in the menu Window| Globals.

Declaration Windows
The windows with the global and module specific declarations are very similar. The list on the **left** contains all declarations, details of the selected entry appear in the window header, and a dump of its

description is given in the **right** list. The module declaration of a variable can be shown with a double click on an **exe token**. In the **left** list you find the offset of the variable, its value, the number of the subroutine that contains a local variable, and its name. At the top of the **module** declarations all Functions in the module are listed, the **global** declarations include the descriptions of all user defined types.

The **names of variables** cannot be copied from the sources to the exe display, because the compiler may reorder or even drop subroutines, but you can **assign** the variable names manually. To do so, you select the variable in the listbox, enter the name in the edit field above and assign it to the variable with the Return key or the Name button.

Changing the **type of a variable** is somewhat more complicated. You select the **scope** (global, module, local...) and the type and assign it with the As button. For arrays, fixed strings, types and objects, **VBMDis** can evaluate the exact type, provided your guess is correct. The **'-'** and **'+'** signs with some types mean that additional informations are stored before or after the location pointed to by the variables offset.

The dump in the right list shows **all** information about the selected variable, so multiple words may be shown even for simple variables. Small arrays with fixed dimensions are often allocated in the description, so these arrays may be initialised in the exe file with an appropriate tool!

Undefined variable types almost come from unused variables and constants, then the Discompiler cannot evaluate its location and type. You can remove all these declarations from your sources, including unused parameters and locals. Unused subroutines almost have a funny parameter list **(p,p,p)**, unused locals and parameters and even function results always have 0000 in the value column.

Global variables and constants are described in the modules by the offset into the global data area. Module constants have the value stored in the module description, whereas ordinary module variables have all zeros there. String constants are represented by pointers into the global string segment.

Storing Informations
In the registered version, all names and types are stored and will be restored whenever you start **VBMDis**. Just like with **VBDis**, you should give each program its own directory, where the descriptions will be stored. If you recompile the program, you should also delete these files or simply the whole directory, else the Discompiler may be fooled by the old and therefore unusable informations from the last analysis.

Custom Controls
In the registered version, **VBCtrl** can be used to analyse custom controls (VBX files) and store the descriptions for use by **VBMDis**.

Each time the Discompiler encounters an unknown custom control, a MsgBox comes up. At this point, start **VBCtrl** and store the description of the control. Then you select Retry in the MsgBox to continue with the description just created. To make this work, **VBMDis** and **VBCtrl** should reside in the same directory, else you must copy the descriptions (*.300) to the directory of **VBMDis**.

VBCtrl - DoDi's Custom Control Tool

VBCtrl analyses VBX files and extracts the properties and events of the included controls. The operation requires some knowledge that I want to impart to you now on the rush.

Start **VBCtrl** and open the desired file from the menu or by double-clicking on it. You can also associate **VBCtrl** with all VBX files and then open a custom control with a double click on the VBX file in the file manager. The file will automatically be analysed, and the **References** window with several lists will be displayed. The list on the **left** ('Symbol to find') shows all possible entry points of the file. Your task is now to find all procedures belonging to the custom controls.

You can recognise the control procedures in most VBXs by the name xxxCTRLPROC (if the VBX developer followed MS' guidelines given in the CDK). Click on the first of these names, and the program displays all references to that procedure in the **middle** list (titled 'Reference to display'). One of these addresses should be the desired control table. If you can't find a corresponding name, or if the supposed tables can't be interpreted properly, try all (even unnamed) entry points one after another, and somewhere you should also find what you're looking for.

Now choose a reference address from the **middle** list to display the supposed table in the list on the **right**. In case of error messages popping up, it is very **unlikely** that you have found a control table. Pay attention to the Class Names (CN) and the Default Control Names (DN), here should appear "readable" text. The first entry (V) allows you to determine the VB Version (100=VB1.0, 200=VB2.0, 300=VB3.0). For details about the other entries shown, please consult the CDK; each line of the list corresponds to one entry in the Control Model structure, the characters on the beginning of each line are abbreviations for the corresponding elements of that structure. If you find multiple tables with identical Class Names (CN), choose the table with the highest Version number.

Pressing the 'Load as Control' button will collect all information related to this control and display it in the **Control Catalogue** window. There you can examine all properties and events, pressing the Help button calls WinHelp to show information concerning the selected event. Then close this window with the OK button, and look for more controls in the **References** window.

If you finally found **all** controls of a VBX and added them to the Control Catalogue, you can **save** the definitions and close the **References** window with the OK button. Doing so, you're asked whether to save the definitions, but
the definitions are really saved only in the registered version of VBCtrl.

The definitions are written to a file named like the VBX, with the extension '300'. This file will be used by **VBMDis** to handle all parts of a program referring to this VBX. The format of the descriptions was reviewed to need less space than before, but only **VBMDis** can handle this new format. You should **never** overwrite the detailed descriptions shipped with the Discompiler, they are needed by both **VBMDis** and **VBDis**!