VBOpt4 - DoDi's Disassembler for VB4

(October 1996)

After a lot of questions about a decompiler for VB4, I sat down and made a first rough version in form of a disassembler, similar to **VBOpt3**. In the moment only a display of the tokens is available, it's a long way to a real Discompiler like **VBDis3**. Nevertheless you'll find many astonishing things about the code produced by VB4, giving hints on how to optimize your programs.

The next step will provide a decompiler for setup programs. I hate that kind of programs, and want to know what they do **before** they can damage my system.

The **demo version** of **VBOpt4** does **not** save any tables created or modified.
Some features described below are also unavailable.

Using VBOpt4

Start **VBOpt4** and open a make file (*.MAK, *.VBP) with Files|*.MAK from the menu.

The **Scan** window pops up and shows what the program is doing.

Then you're asked for the **project directory**. The demo version creates only one file with the preprocessed sourcecode, but the registered versions create many more files. It will be easier to remove these files if you use a dedicated subdirectory as project directory.

If the project contains informations about both 16 and 32 bit versions, you're asked to specifiy the desired version.

Then the sources are preprocessed, resulting in a single text file with only the lines that belong to the selected version (16/32 bit). This file may be useful if you want to know which parts of your sources are really compiled.

The preprocessor of **VBOpt4** is optimized for speed, so it may be necessary that you enter the value of constants or expressions, that the preprocessor cannot evaluate itself.

After the sources are scanned, **VBOpt4** compares them with the executable file, to verify that the declarations are valid for that program. If not, or if the program exceeds some limits of the demo version, a message is displayed and the program terminated.

If no errors occured so far, two windows appear in the main window, with your sources in the **Source** window and the tokens of the compiled program in the **Exe** window.

Then you can choose a module and a section from any of these windows. The **Sync** button selects and displays the same section as currently shown in the other window.

Depending on the resolution and size of your screen it may be difficult to read text shown with a monospaced font. In this case you can select View | Fixed Font from the menu and set the size and attributes of that font as desired.

Window Overview

The **Main** window contains the **Source** window with your source code and the **Exe** window with the tokens of the compiled program. From the menu you can open more windows:

Window	Contents
About	version and copyright of the program
Assembler	all defined assembly instructions
Classes	all OLE classes found in the program and the standard DLLs of VB4.
Controls	the custom controls built into **VBOpt4**
Globals	the global variables of the program
Locals	the local variables of the current subroutine
References	all locations that reference the selected location in the exe file
Resource	the resources of the exe file
Scan	the progress of scanning the sources and the exe file
Segments	dumps of all locations found in the segments of the exe file
Statics	the static variables in subroutines of the current module
Tables	all general tables with informations about the program
Text	any text file (in the Main window)
Tokens	all defined exe tokens
Types	the user defined types (UDT) found in the source code

Most windows need no special description (or are unavailable in the demo version).

The Token Display

The **Exe** window shows the offset of the tokens relative to the start of the subroutine, a description (if the meaning of the token is roughly known), and the arguments.

Some lines containing "**newline**" (or something equivalent) are inserted in places where the end of a statement is assumed, but not guaranteed. You can create reliable marks for the statements by putting an "**On Error**" statement into your source code, forcing VB4 to compile "**statement**" tokens for every statement.

Lines with **line numbers** are always flagged correctly, with possible multiple occurences of the same number, due to a bug in some VB4 compiler.

Most tokens found in VB4 programs are displayed with a description. Unknown tokens or tokens with a wrong description of it's arguments switch the display to a hex dump until the end of the subroutine. In this case the token in error is displayed in the **Token** window, where you can enter a better description.

You can also examine and change the description of a token with a double click on the corresponding line in the **Exe** window.

The arguments of the tokens can be displayed as variable names, whenever a declared variable is accessed. You can use the check box above the token display to enable or disable the display of the variable names (not in the demo version).

Other buttons in the **Source** and **Exe** windows are:

Sync shows the same subroutine as is currently displayed in the other window.

The Token Window

The **Token** window displays the descriptions of all known tokens, with the hex code, verbose description and arguments. To **add** a token, enter it's **code** and **argument** string (the description is optional), and press Modify. You can scroll through the descriptions with the Spin button, **edit** a description and confirm the changes with the Modify button. With the Save button the modified descriptions are immediately saved to disk.

Then press the Refresh button to see the updated display in the **Exe** window.

The **argument string** can consist of the following characters (lower case for 1 word each, upper case for 2 words):

~	means a variable number of arguments, stored in the word following the token code. It can appear only as the **first** character in the argument string.

In a variable list of arguments these characters are allowed after "~":

L	the arguments are all local variables
D	the arguments are all displacements (labels with On GoSub/GoTo...)
$	the last argument is a string literal, starting with the character count
u	the last argument is a Unicode literal, starting with the character count
g	a description of the UDT in a Get or Put statement

These characters are always allowed:

w	is an (unknown) word.
P	is a pointer (VB4 compiles pointers to subroutines and some data structures). You must use this argument type if the dump shows a pointer like "1:2345" or a symbol name instead of a simple four digit hex word.
p	starts a specific pointer declaration, and must be followed by a type character. These descriptions will be used later, currently defined are:
pc	pointer to a GUID, that evaluates to a class name.
pz	declares a pointer to a zero terminated string, that is displayed as a comment after the pointer.
d	shows a displacement as absolute offset (in 'jump' tokens).
e	is the argument to an "On Error" token.
l	displays the name of a parameter or local variable.
M	displays the name of a global variable (2 words).
m	displays the name of a property or method.
s	displays the name of a static variable (not always appropriate)
% & ! # @	displays arguments formatted according to the VB type character.
4	dumps a 4 byte constant (Long, Single)
8	dumps a 8 byte constant (Currency, Double, Date)

The last character can be:

_	inserts a "newline" comment after the token

Please note that some tokens can have different constant arguments, e.g. # and @. In this case using a special type may result in a wrong display, if the argument really has the other type. In this case, use 4 or 8 to show the argument without formatting.

Some more argument types may be introduced, for references to local and global variables, descriptions of objects and other data structures.

To clarify the operation of the tokens and the type of variables, the following **additional type characters** are used:

°	Byte
²	Boolean
³ or *	String * <fixed length>
~	Variant
/	Date
{	User Defined Type (like a struct in C)
.	any object (with properties and methods)
4 or |	4 byte operand like & and ! (or a pointer)
8	8 byte operand like @ and #
^	is sometimes used to denote a pointer or reference

A more systematic description of the operation of the tokens will be provided later. Sometimes "t" or "vt" is used if a token is assumed to deal with temporary variables, invisible in the source code.

Tokens in VB3 programs were found in the range &H0F00 to &H4B00, all values outside this range (at least >= &h8000) should be considered as arguments of the preceding token. Values up to &h7FFF may be tokens new to VB4. VB4/32 seems to have a single jump table for all tokens, they are contiguously numbered in steps of 2. Moreover, most tokens with the same meaning, but different argument types, are assigned sequential codes. This is similar to VB3, where the tokens for variables with an appended type character are also grouped together, but in VB4/32 the tokens in such a cluster are really different, eg. CVar(%), CVar(&)...

Some Background Information

When you open a project for the first time, an **extensive scan of the whole project** is executed. During this time the **Scan** window is displayed, showing the progress of the analysis.
The tables created from this step are saved to files and used when you reopen the same project later (not in the demo version).

The first action is to **preprocess all conditionals** (#If...) in the sources. The preprocessed text is saved to a ***.T0** file, where * is the project name. Constants are not replaced in this step, to keep the source text as close to the original as possible. Predefined are the constants **True** and **False**, as well as **Win16** and **Win32**, depending on the exe type. Constants found in the make file are added to this list. Expressions in **#If** and **#ElseIf** statements may evaluate to wrong results, currently no operator precedence is implemented.

Therefore you should use not more than one "And" or "Or" in conditional expressions.

Then a map of all sections in the source files is created (e.g. forms, declarations, subroutines), used to display the source modules in the **Source** window.

Descriptions for all **User Defined Types** are created, used to determine the locations of local variables in the subroutines, and the member names when a Type variable is accessed.

Then the exe file is scanned, resulting in a complete list of all modules, subroutines and other elements.

The **Forms** found in the exe file are displayed in text format, ready for usage with VB4. Some sizes may be shown incorrectly, I hope this can be fixed in some later version.
VBDis3 can handle any custom controls, when a description (*.300) is provided, but cannot display the forms in text format. Therefore all forms must be converted to text format using VB3, sometimes a lengty operation. This was changed in **VBOpt4**, now the forms can be output directly in text format, with the consequence that only the fully known controls built into the VB runtime DLL can be handled.

Constants are never stored in specific locations, their values are displayed on every occurence in the **Exe** window.

Then the result is displayed in the **Source** and **Exe** windows. When you select a subroutine in the **Source** window, the locations of the local variables are determined. The offsets calculated in this step are used to display the names of the variables in the **Exe** window. Incorrect calculations will result from declarations containing Constant names instead of numeric values, as in "var As String * someConstant" or "var(low To high)". In this case VBOpt4 asks you to enter the correct value of the expression.

Therefore you should replace all constants in such expressions by their numeric value before using VBOpt4.

A lot of temporary variables are displayed in most subroutines. You can determine such variables by offsets below the last local variable shown in the local variables list. The search for local variables in the source files is terminated with the first executable statement, because this can result in temporary variables inserted by the compiler, before the next Dim statement.

Therefore you must place all Dim statements at the begin of a subroutine, to obtain a complete list of the local variables.

The primary goal of the future **VBDis4** is to decompile setup programs. Therefore I use SETUP1.VBP from the VB4 setupkit directory to test and improve the decompiler. You can

compile this project and check the capabilities of **VBOpt4** with it, too.

Current Limitations

If your **project** sources **differ** from the compiled program, **VBOpt4** is terminated. The **demo** versions have more restrictions, in the maximum number of forms, variables...

Different versions of VB4 may have different allocations strategies for variables. At least the TR6 beta creates much more code and uses many more temporary variables than the following releases. This may result in incorrect variable names shown in the **Exe** window.

VBOpt4 may crash with out-of-memory and subscript errors if a project is too big. If this occurs, I'll introduce range checks in the demo version to prevent such crashes, and the registered versions will be updated to overcome such limits.

Constants are **not** evaluated in **Dim** statements, you can enter the appropriate value in an InputBox.

Local variables can be handled only if the declarations occur at the begin of a subroutine. Intermediate statements may result in temporary variables allocated by the compiler, moving the declared variables to some unknown location, depending on the version of the compiler.

Dynamic arrays created with **ReDim** are not handled now.

Class modules and **Property functions** are not handled yet.

Objects, methods and properties are partially handled.

Only the **controls** built into the VB4 runtime DLL can be handled now.

The forms shown contain some wrong sizes.

History of VBOpt4

ToDo List

- Handling constants in Dim statements.

- Correct sizes in the form descriptions.

- First version of a Discompiler.

- Tracking pointers through multiple dereference steps.

Version 4.05 from October 1996

- Handling of both 16 and 32 bit programs.

- Display of control names, methods and properties.

- Correct font size in forms.

Version 4.16.04 from June 1996

- Evaluation of class references, needed to show properties and methods. The information about classes are derived from TYPELIB informations, found in DLLs, OLBs and TLBs.

- The data structures found in 16- and 32-bit executables are very similar, so the same strategies can be used for both kinds, though the different structure of the exe files themselves may be better handled in different programs, with a common front end that detects the type of the program and invokes the appropriate back end.

Version 4.16.03 from April 1996

- Modules, subroutines and variables are automatically matched between the executables and the sources. The **Subroutines** window and some buttons related to the manual assignment were removed.

- All variables are located on the first pass through the source text, the display of variable names is now independent from the source code currently shown.

- References to variables are always displayed with their name found in the source code.

- Windows were added to display the global, static and local variables.

- A single file is used for the preprocessed source code.

Version 4.16.02 from March 1996

The first published version for VB4 (16 bit).

First Conclusions on VB4

VB4 does **not** compile the source code into the executables.
Statement separators are created only if needed for error handling, forced with "**On Error**".
"**Debug.Print**" statements are not compiled into the executable, but "**Stop**" is.

VB4 removes redundant type conversions, e.g. **CInt** from "i1%=CInt(i2%)", but preserves other conversions, e.g. **CVar** in "s1$=CVar(s2$)", resulting in **CVar** followed by **CStr**.

No general descriptions for **user defined types** were found yet, only descriptors for Strings and Variants (possibly containing strings). The members of an UDT are accessed directly by their offset in the data structure.
Only in every **Get** and **Put** statement a lengthy description of variable UDTs is included.

Constants are stored as tokens with immediate arguments, not as variables.
I suppose this comes from the preprocessor of VB4, that replaces all constants by the value, thus decreasing the symbol table size of the compiler. Unfortunately the preprocessor doesn't evaluate constant expressions, resulting in superfluous operations at runtime.
Therefore you cannot patch a constant value in a single place, as is possible with VB3.
To encrypt and minimize the size of your program, replace string constants by variables and initialize them in the start procedure of the program.

Forms are stored in a similar way as VB3 did, but now directly **including the names** of all controls.
The names of the controls cannot be removed, because the controls are referenced by their name, at least if variables of type Form are used. The same applies to Control variables and their properties and methods, and possibly to any OCX.

Where VB3 createt different tokens for the same function, VB4 now uses the **same token** for different tasks, e.g. "jz" and "jnz" for If, Else, Do, While, Select...

VB4 calls every subroutine adding at least one **invisible argument** for a possible return value.
Methods are always called with a Variant return argument, that is not used if the method doesn't return a value.
Procedures in **forms** are possibly called with a Form object. This applies to every subroutine in a form, not only to event procedures. **Classes** will have a similar implementation (the **this** pointer in C++).

VB4/16 can compile the routines of a module into different segments, based on the code size of the routines.

VB4 creates some native **inline code** for subroutine calls, and true pointers through fixups.
Though this may speed up execution, everybody can use a debugger, put breakpoints on any subroutine call and inspect or modify the parameters.
Therefore you should never pass textual passwords to a subroutine, even in calls to subroutines inside your program. Better use a common variable, or encrypt the password before passing it to an external subroutine.

My Suggestions for the Implementation of VB5

To reduce the size of a program, these elements should be kept in distinct locations:

- constant strings
- descriptions for UDTs with variable arguments (strings, variants) for Get/Put

To increase the execution speed of a program,

- **references** to properties and methods of controls should be evaluated once, when the object is created. This applies especially to forms and OLE objects. A similar approach as used in VB3 (early binding for Form and Control variables) could be used to cover all fixed references to unspecific objects.

- **methods** without return values should be called without a return argument, thus eliminating the initialization and destruction of the unused variant.

- **variants** should be used only where absolutely required, **never** as dummy arguments that are not used in subroutine calls.

To reduce memory usage,

- **strings** should be separated explicitly in Unicode (16 bit) and Ansi (8 bit). This also reduces type conversions in calls to external subroutines, if the character set is already available as required.

- **Boolean arrays** should be implemented as **bit** arrays. There is no other obvious usage of the **Boolean** type, in comparison to **Integer**.

- **constant expressions** should be evaluated at compile time

- **unused temporary variables** should be removed

To improve program development,

- **type checking** should be done during compilation as far as possible. Using **Variant** variables is very convenient to compiler writers (and decompiler writers <gd&r>), but prohibits type checking at compile time.

- assumptions on the type of properties, arguments and return values of methods should be classified at least as numeric, text and other known or unknown data structures.

- it should be up to the programmer to use **Variant** variables during the first development step of a program, and replace them by specific types later. Using literal constants for variable types (like **typedef** in C) is highly desirable.

Extensive type checking can reveal many errors at compile time and remove many type conversions at runtime. In addition, error handling code in a program will be smaller or even unnecessary if the return type of methods and properties is known.

The need of separate tools like Unix **lint** to find obvious type mismatches is inadequate to modern compiler technology.

The **STRICT** option in the Windows header files and the many (often wrong and almost superfluous) **typecasts** in the SDK samples show what features are missing in Microsoft's C

compilers; a VB compiler should not need such crutches.

On the other hand, the source code of the setup1 project shows that the programmers around VB are not very good. The usage of **IIf** or explicit **If/Then/Else** statements to convert the result of a comparison operation and assign it to a Boolean variable obviously shows, that the author either doesn't know about **True** and **False** and the handling of boolean values in Basic, **or that he doesn't trust in the implementation of VB**.

What comes next?

In the beginning it was not clear whether a decompiler for VB4 programs is feasable. Now I'm quite sure that it's possible. The only question is, whether it's **desireable to publish** such a tool or not. For VB3 it was necessary to publish **VBDis3**, forcing Microsoft to remove the source code from the programs compiled with VB4. On the other hand, **VBDis3** could definitely help a lot of programmers to recover lost sources!

As you may already know, I hate setup programs that corrupt the INI files, autoexec.bat and config.sys of my system and overwrite DLLs, VBXs and OCXs with old versions or in different languages. Therefore I promised to publish at least a Discompiler for setup programs - and this one will come!

But before I can publish it, I must find a way to protect programs against misuse of that decompiler. It's not very easy to find data structures in VB4 programs that can be modified to confuse or crash a decompiler, but keep the program running. It were a poor solution to overwrite some bytes, just like putting in a prayer "This program shall not be decompiled". But I hope to find a more practical solution (size of a program, number of forms) to prevent decompilation of other programs, **without the need for explicit protection of already existing VB4 executables**.

A handy restriction is a new feature built into **VBOpt4**: where the user had to convert all the forms created by **VBDis3** from binary to text format, **VBDis4** will create all modules in one pass as text. Therefore only the controls built into **VBDis4** can be used in a program, else it cannot be decompiled.

VBOpt4 is restricted to programs containing at most 8 forms and only the custom controls used in the setup1 program of the VB4 setup kit.

A decompiler will come, however, to recover lost source code. Every decompiler of this kind will be restricted to a specific program, so you need not fear about the privacy of your VB4 sources.

In addition, if the installation of a Windows application doesn't become much more transparent to the user, more decompilers will come for future versions of VB and MS(V)C, and these will be able to decompile <u>at least</u> setup programs.

A decompiler for C/C++ is a big task, so it may become a public project, with the sources available to all programmers willing to assist in developing a general decompiler for 80x86 processors, as I made myself already for 680x0 processors. As can be seen with VB4 programs, most code of a Windows application is contained in DLLs and common libraries like MFC or OWL. Once these parts are identified in a program, they mustn't be decompiled because they are already documented. Even better, every common subroutine allows to locate and type the data structures it uses, resulting in many useful informations about the rest of the program. This strategy was successfully tested for several compilers on Atari and Amiga systems, as well as on Unix workstations.

DoDi
(Dr. H.-P. Diettrich)

CompuServe: 100752,3277
Internet: 100752.3277@compuserve.com
WWW: http://ourworld.compuserve.com/homepages/DoDi