OllyDbg 2.01 Brief Help

Table of contents

 *© squashed bug 2.0*

A journey of a thousand li begins with a Single Step
- Chinese proverb

Introduction

Differences between versions 1.10 and 2.xx

The second version of the 32-bit debugger OllyDbg is redesigned practically from the scratch. As a result, it is faster, more powerful and much more reliable than its predecessor. Well, at least in the future, because some useful features from the version 1.10 are not yet available in 2.01.

Version 2.01 contains many new features that were not available in 1.10. Among them:

- Full support for SSE and AVX instructions. SSE registers are accessed directly, without code injection;
- Execution of commands in the context of debugger, allowing run trace speed - with conditions and protocolling! - of up to 1,000,000 commands per second;
- Unlimited number of memory breakpoints;
- Conditional memory and hardware breakpoints;
- Reliable, analysis-independent hit trace;
- Analyser that recognizes the number (and sometimes the meaning) of the arguments of unknown functions;
- Detaching from debugged process;
- Debugging of child processes;
- Option to pause on TLS callback;
- Option to pass unprocessed exceptions to the unhandled exception filter;
- Built-in help for integer and FPU commands;
- Customizable shortcuts;
- Support for multiple languages in user interface.

OllyDbg 2.01 overview

OllyDbg 2.01 is a 32-bit assembler-level analyzing Debugger with intuitive interface. It is especially useful if source code is not available or when you experience problems with your compiler.

Requirements. Developed and tested mainly under Windows XP, but should work under any 32-bit Windows version: NT, 2000, XP, 2003 Server, Vista, Windows 7 and so on. Old DOS-based versions (95, 98, ME) are no longer supported. That is, if you install Microsoft Layer for UNICODE, you will be able to start OllyDbg and even set breakpoints, but I can't promise stable operation. Version 2.01 will not work under 64-bit Windows yet! For a comfortable debugging you will need at least 1-GHz processor. OllyDbg is memory hungry. If you debug large application with all features enabled, it may easily allocate 200-300 megabytes for backup and analysis data.

Supported instruction sets. OllyDbg 2.01 supports all existing 80x86-compatible CPUs: MMX, 3DNow!, including Athlon extensions, SSE instructions up to SSSE3 and SSE4, and AVX instructions.

Configurability. More than 120 options (oh, no! This time it's definitely too much!) control OllyDbg's behaviour and appearance.

Data formats. Dump windows display data in all common formats: hexadecimal, ASCII, multibyte, UNICODE, 16- and 32-bit signed/unsigned/hexadecimal integers, 32/64/80-bit floats, addresses, disassembly (MASM, IDEAL, HLA or AT&T). It also decodes and comments many Windows-specific structures, including PE headers, PEBs, Thread data blocks and so on. You can dump system memory (XP only), files and raw disks.

Help. OllyDbg 2.01 includes built-in help on all 80x86 integer and floating-point commands. If you possess Windows API help (*win32.hlp*, not included due to copyright reasons), you can attach it and get instant help on system API calls.

Startup. You can specify executable file in command line, select it from menu, drag-and-drop file to OllyDbg, restart last debugged program or attach to already running application. OllyDbg supports just-in-time debugging and debugging of child processes. You can detach from the debugged process, and it will continue execution. Installation is not necessary!

Code highlighting. Disassembler can highlight different types of commands (jumps, conditional jumps, pushes and pops, calls, returns, privileged and invalid) and different operands (general, FPU/SSE or segment/system registers, memory operands on stack or in other memory, constants). You can create custom highlighting schemes.

Threads. OllyDbg can debug multithread applications. You can switch from one thread to another, suspend, resume and kill threads or change their priorities. Threads window displays errors for each thread (as returned by call to GetLastError).

Analysis. Analyzer is one of the most significant parts of OllyDbg. It recognizes procedures, loops, switches, tables, GUIDs, constants and strings embedded in code, tricky constructs, calls to API functions, number of function's arguments, import sections and so on. It attempts to determine not only the number of stack arguments in the unknown functions, but even their meaning. Analysis makes binary code much more readable, facilitates debugging and reduces probability of misinterpretations and crashes. It is not compiler-oriented and works equally good with any PE program.

Full UNICODE support. All operations available for ASCII strings are also available for UNICODE, and vice versa. OllyDbg is able to recognize UTF-8 strings.

Names. OllyDbg knows symbolic names of many (currently 10800) constants, like window messages, error codes or bit fields, and decodes them in calls to known functions.

Known functions. OllyDbg knows more than 2300 frequently used Windows API functions and decodes their arguments. You can add your own descriptions. You may set logging breakpoint on a known or guessed function and protocol arguments to the log.

Calls. OllyDbg can backtrace nested calls on the stack even when debugging information is unavailable and procedures use non-standard prologs and epilogs.

Stack. In the Stack window, OllyDbg uses heuristics to recognize return addresses and stack frames. If program is paused on the known function, stack window decodes arguments of known and guessed functions. Stack also traces and displays the chain of SE handlers. If integrated stack walk fails, you may switch to *Dbghelp.dll*.

Search. Plenty of possibilities! Search for command (exact or imprecise) or sequence of commands, for constant, binary or text string (not necessarily contiguous), for all commands that reference address, constant or address range, for all jumps to selected location, for all referenced text strings, for all intermodular calls, for masked binary sequence in the whole allocated memory, for integer or floating-point number and so on. If multiple locations are found, you can quickly navigate between them.

Breakpoints. OllyDbg supports all common kinds of breakpoints: soft (INT3 or several other commands),

memory and hardware. You may specify number of passes and set conditions for pause. Breakpoints may conditionally protocol data to the log. Number of soft and memory breakpoints is unlimited: in the extreme case of hit trace, OllyDbg may set hundreds of thousands INT3 breakpoints. On a fast CPU, OllyDbg can process up to 20-30 thousand breakpoints per second.

Watches. Watch is an expression evaluated each time the program pauses. You can use registers, constants, address expressions, boolean and algebraic operations of any complexity.

Execution. You can execute program step-by-step, either entering subroutines or executing them at once. You can run program till next return, to the specified location, or backtrace it from the deeply nested system API call back to the user code. When application runs, you keep full control over it. For example, you can view memory, set breakpoints and even modify code "on-the-fly". At any time, you can pause or restart the debugged program.

Hit trace. Hit trace shows which commands or procedures were executed so far, allowing you to test all branches of your code. Hit trace starts from the actual location and sets soft breakpoints on all branches that were not traced so far. The breakpoints are removed when command is reached (hit).

Run trace. Run trace executes program in the step-by-step mode and protocols execution to the large circular buffer. Run trace is fast: when fast command emulation is enabled, OllyDbg traces up to 1 million commands per second! Run trace protocols registers (except for SSE/AVX), flags, contents of accessed memory, thread errors and - for the case that your code is self-modifying - the original commands. You may specify the condition to stop run trace, like address range, expression or command. You can save run trace to the file and compare two independent runs. Run trace allows to backtrack and analyse history of execution in details, millions and millions of commands.

Profiling. Profiler calculates how many times some instruction is listed in the run trace buffer. With profiler, you know which part of the code takes most of execution time.

Patching. Built-in assembler automatically selects the shortest possible code. Binary editor shows data simultaneously in ASCII, UNICODE and hexadecimal form. Old good copy-and-paste is also available. Automatic backup allows to undo changes. You can copy modifications directly to executable file, OllyDbg will even adjust fixups.

UDD. OllyDbg saves all program- and module-related information to the individual file and restores it when module is reloaded. This information includes labels, comments, breakpoints, watches, analysis data, conditions and so on.

UDL. You may convert standard libraries supplied with your compiler to the UDL library and tell Analyser to recognize library functions in the code.

Plugins. You may extend the functionality of the OllyDbg 2.01 by writing your own plugins (or downloading plugins from the Internet).

Customization. You can specify custom fonts, colour and highlighting schemes.

Text-to-speech. For handicapped users: activate text-to-speech option, and OllyDbg will read the selections.

And much more! This list is far from complete, there are many features that make OllyDbg 2.01 the friendly debugger.

(No) registration

 It is a closed-source freeware. For you as a user this means that you can use OllyDbg for free, whether for private purposes or commercially, according to the license agreement. Registration is not necessary. I have introduced it for the first version just to see

how popular my program is. The results were above any expectations. I was overwhelmed with the registration forms, and this had noticeable negative influence on my productivity. Therefore: there is **no registration** for the OllyDbg 2.01.

If you are professor or teacher and use OllyDbg for **educational purposes**, I would be very glad if you contact me (Ollydbg@t-online.de), especially if you have ideas how to make my product more student-friendly or better suited for the educational process.

Legal part

Trademark information. All brand names and product names used in OllyDbg, accompanying files or in this help are trademarks, registered trademarks, or trade names of their respective holders. They are used for identification purposes only.

License Agreement. This License Agreement ("Agreement") accompanies the OllyDbg version 2.01 and related files ("Software"). By using the Software, you agree to be bound by all of the terms and conditions of the Agreement.

The Software is owned by Oleh Yuschuk ("Author") and is Copyright (c) 2000-2013 Oleh Yuschuk. You are allowed to use this software for free. You may install the Software on any number of storage devices, like hard disks, memory sticks etc. and are allowed to make any number of backup copies of this Software.

The Software is distributed "as is", without warranty of any kind, expressed or implied, including, but not limited to warranty of fitness for any particular purpose. In no event will the Author be liable to you for any special, incidental, indirect, consequential or any other damages caused by the use, misuse, or the inability to use of the Software, including any lost profits or lost savings, even if Author has been advised of the possibility of such damages. In particular, the Author is not responsible for any damages caused by the use of third-party plugins attached to the Software.

You are not allowed to modify, decompile, disassemble or reverse engineer the Software except and only to the extent that such activity is expressly permitted by applicable law. You are not allowed to distribute or use any parts of the Software separately. You may make and distribute copies of this Software provided that a) the copy contains all files from the original distribution and these files remain unchanged; b) if you distribute any other files together with the Software, they must be clearly marked as such and the conditions of their use cannot be more restrictive than conditions of this Agreement; and c) you collect no fee (except for transport media, like CD), even if your distribution contains additional files.

This Agreement covers only the actual version 2.01 of the OllyDbg. All other versions are covered by similar but separate License Agreements.

Fair use. Many software manufacturers explicitly disallow you any attempts of disassembling, decompilation, reverse engineering or modification of their programs. This restriction also covers all third-party dynamic-link libraries your application may use, including system libraries. If you have any doubts, contact the owner of copyright. The so called „fair use" clause can be misleading. You may want to discuss whether it applies in your case with competent lawyer. Please don't use OllyDbg for illegal purposes!

Your privacy and security. The following statements apply to versions 1.00 - 2.01 at the moment when I upload corresponding archives (containing OllyDbg.exe and support files) to Internet ("original OllyDbg"). They do not apply to any third-party plugins.

I guarantee that original OllyDbg:

- never tries to spy processes other than being debugged, or acts as a network client or server, or sends any data to any other computer by any means (except for remote files specified by user), or acts as a Trojan horse of any kind, with one exception: if you allow *dbghelp.dll* access to the

Microsoft Symbol Server, this DLL may exchange information with the Microsoft. The corresponding option is turned off by default and you must activate it explicitly;

- neither reads nor modifies the system Registry unless explicitly requested, and these requested modifications are limited to the following two keys:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug\Debugger
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug\Auto

- does not create, rewrite or modify any files in system directories without your explicit request;

- does not modify, unless explicitly requested, any executable file or DLL on any computer, including OllyDbg itself;

- logs your debugging activities only on your explicit request (except for the File History kept in *ollydbg.ini* and *.udd files with debug information). Even more, I guarantee that without your allowance OllyDbg will create or modify files only in the directory where it resides and in the working directories specified in the Options dialog;

- (last but by no means least) contains no apparent or hidden „nag" screens forcing you to register this program, nor any features that limit the functionality of the OllyDbg depending on the registration or after any period of time.

Beware of viruses. Although I've checked the original archive with several virus scanners, please do not assume that your distribution of OllyDbg is free from viruses or Trojan horses camouflaged as OllyDbg or support routine. I accept no responsibility for damages of any kind caused to your computer(s) by any virus or Trojan horse attached to any of the files included into archive, or to archive as a whole, or for damages resulting from the modifications applied to OllyDbg by third persons.

Installation

OllyDbg requires no installation. Simply create new folder and unpack *Odbg201.zip* to this folder. If necessary, drag-and-drop link to *ollydbg.exe* to the desktop to create shortcut. Under Windows 7, open shortcut properties and activate "Run as administrator".

If you are a hardcore user and run OllyDbg on Windows NT 4.0 (no problems!), you will need *psapi.dll*. This library is not included.

Version 2.01 is not intended to work on DOS-based Windows versions, but this is possible provided you install MIcrosoft Layer for UNICODE. Some very, very old versions of Windows 95 do not include API functions *VirtualQueryEx* and *VirtualProtectEx*. These functions are very important for debugging. If OllyDbg reports that functions are absent, normal debugging is not possible. Please upgrade your OS. Anyway, Windows 95, 98 and ME are no longer supported. Use OllyDbg 1.10 It's free and compatible with these systems.

Support

The available support is limited to the Internet page http://www.ollydbg.de. From here you will be able also to download bugfixes and new versions. If you have problems, send email to Ollydbg@t-online.de. Usually I answer within a week. **Unless you explicitly disallow this, I reserve the right to place excerpts from your emails on my Internet site**.

Full source code is available but will cost you money. This is a „clean-room" implementation that contains no third-party code. You can order either the whole source code or its parts like Disassembler, Assembler or Analyzer. To get more information, send me a mail. By the way, Disassembler 2.01 is released under GPL v3, you can download it from my page.

Description of .udd file format for any OllyDbg version is available on request. It is free.

Settings used in this help

There are many options that insfluence the appearance and the functioning of the OllyDbg. This help file assumes that options are in their default state.

This also includes shortcuts. Most OllyDbg shortcuts can be redefined. If you have installed different shortcuts, either use menu, or restore default shortcuts (**Options | Edit shortcuts... | Restore defaults**).

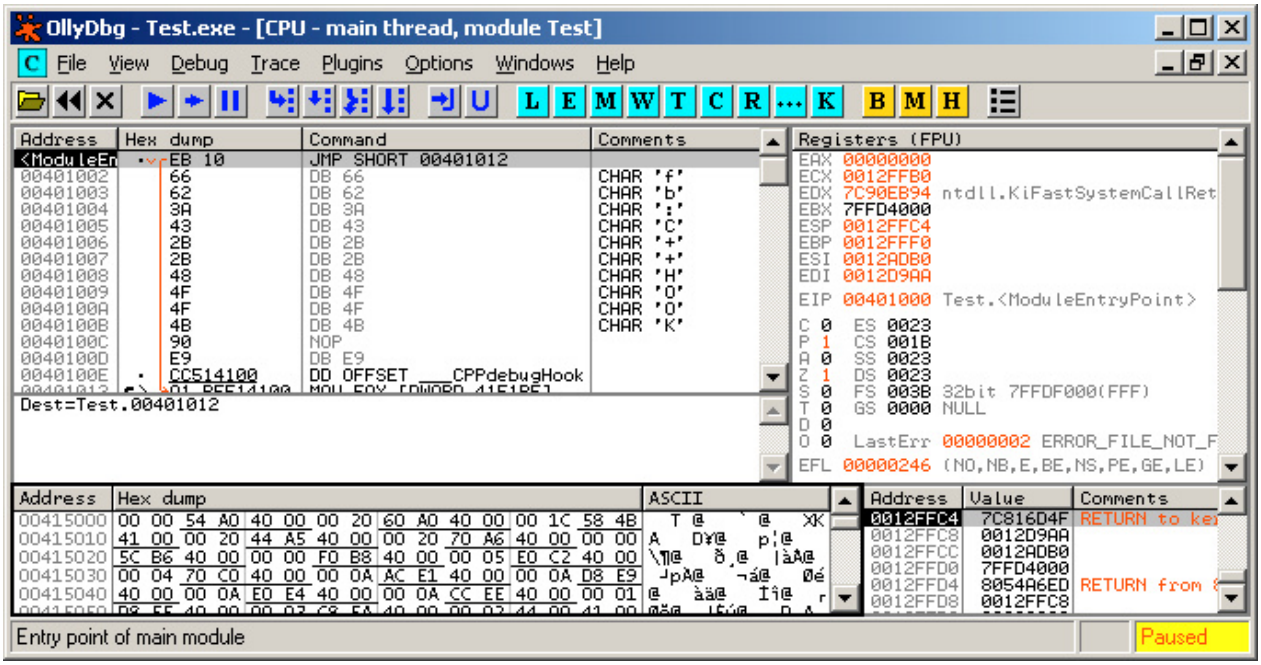
First steps

Starting an application

The best way to get familiar with a new program is by making simple exercises. I assume that you are briefly familiar with the architecture and command set of the 80x86 processor. Also you understand the concepts of process, thread and module and know that Portable Executable file consists of headers and sections.

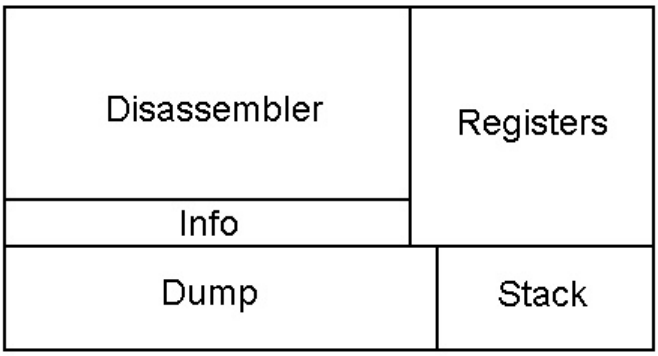
There is a small test application in the OllyDbg distribution archive named - surprise! - *Test.exe*.

Start OllyDbg and load *Test.exe*. You may, for example, drag and drop this test application from Explorer, or press **F3**, or choose **File | Open** from the main menu. OllyDbg will open application, analyse its code and pause at the entry point of the main module. (Main module is the *Test.exe* itself):



What you will see depends on many options that control OllyDbg's behaviour, but your picture will be similar to the one above. OllyDbg is an old-style MDI application without graphical bells and whistles. Its appearance is optimized for performance. Small default fonts and narrow borders maximize the amount of information you can see on the display. Of course, you can always adjust colours and fonts so that they best suit your eyes.

Most of the OllyDbg's screen above is filled with the CPU window. This is the window where you will spend most of your time debugging the application. It consists of five panes:



Disassembler lists commands in the selected piece of memory. For each command, information includes hexadecimal address in memory and its label, binary command code, command disassembled to text and comments associated with the command.

If application is paused, current execution point (where register EIP points to) is marked black in the Address column. Unconditional breakpoints are red and conditional have magenta background.

Most comments are created by Disassembler and Analyser. They help you to understand the meaning of the code. You may add your own comments, too.

If you are the beginner, select command and press Ctrl+F1 to get help on this command. (Currently only integer and FPU commands are supported).

Info pane displays additional information about the first selected command. It may include the contents of operands and their decoding, list of known jumps and calls to the command, for conditional commands - whether jump is taken, if available - source line, loop variables if command belongs to the recognized loop, and more.

Registers pane displays the contents of registers for the selected thread. Letters C, P, A and so on stay for the individual CPU flags (Carry, Parity, Auxiliary Parity...). The flags are parts of the Flag register EFL. Field Last err is not a register, it displays the contents of memory where Windows API functions store error codes. For example, if call to *CreateFile("abc.def",...,OPEN_EXISTING...)* was unable to find the file *abc.def*, it will set last error to 0x00000246 (ERROR_FILE_NOT_FOUND).

The contents of register or bit is highlighted if it was changed since the last pause or if it was modified by the user.

FPU registers are shared between FPU, MMX and (in the case of AMD) 3DNow! command sets. To change their presentation, use popup menu or press bar on the top of the pane.

Stack pane shows stack associated with the selected thread. Address in ESP is marked black in the Address column. OllyDbg attempts to recognize stack frames and marks them with parentheses to the left from the contents. It also highlights doublewords that may represent possible returns. But care - these may be the remnants of the previous calls! A condensed view of the stack is available in the Stack window.

Dump initially displays the data area of the main module. You may select any location and many different data presentations here. Dump is thread-independent.

The status bar at the bottom shows debugging messages and displays current state of the debugging. Red text "Paused" in the right bottom corner means that application is suspended by the OllyDbg. Another possible states are "Loading", "Running", "Tracing", "Terminated" etc.

Lesson 1 - Breakpoints

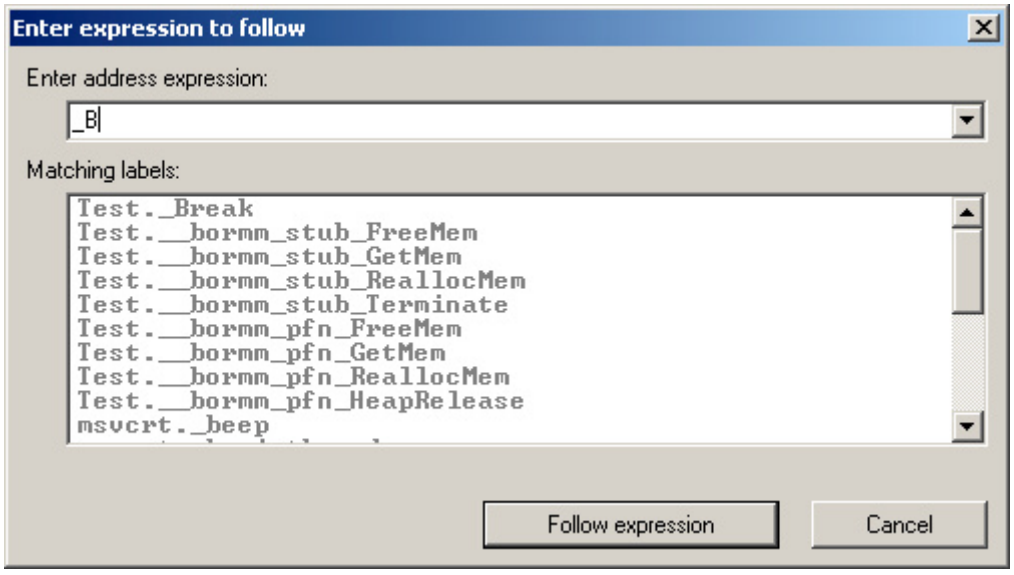
In their simplest form, breakpoints are requests to pause execution when some address is reached or accessed. OllyDbg also supports conditional breakpoints (will pause execution only if some condition is fulfilled, like ECX==1) and logging breakpoints (they protocol some data to the log without stopping the program).

There are three different types of breakpoints: software, hardware and memory. To set soft breakpoint, debugger replaces first byte of the command with another command that will cause exception or interrupt when it executes. Usually this is a dedicated command INT3 (binary code 0xCC), but other privileged commands, like HLT or CLI, can also be used instead.

Load *Test.exe* into the OllyDbg as described above. Status will change to "Paused". (If not, open **Options | Start**, change first pause to **Entry point of main module** and restart). Select **Debug | Run** from the main menu, or press **F9**, or press button with the right triangle  . Status will change to "Running" and you will see the main screen of the test application.

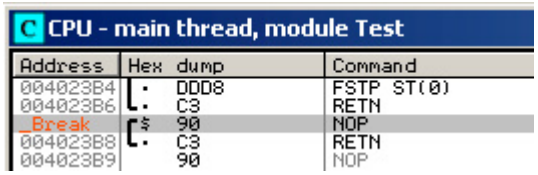
Now press button "**Read [_Break]**". You will see the message "**[Byte _Break] = 0x90 (NOP)**". When you press this button, test application reads byte at symbolic address *_Break* and displays its contents.

Change focus to Disassembler pane by clicking somewhere inside it. Have you noticed the thin black contour around the pane? Now press **Ctrl+G** (or right click and select **Go to | Expression...** from the pop-up menu) and start typing "_Break":



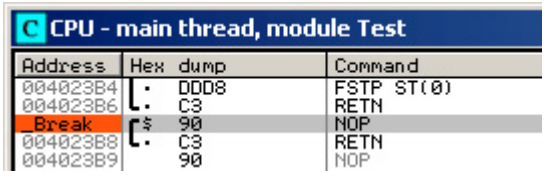
The appearing window allows you to change address of the memory displayed in the Disassembler window. You may use direct addresses, like 401023 (hexadecimal address 0x00401023), symbolic labels like *_Break* or expressions like ECX+EDX*4. Note that all constants are assumed to be hexadecimal. If you want to specify decimal number, follow it with a decimal point: 100 means $0x100_{16} = 256_{10}$, and 100. means 100_{10}. (And 100.0 means floating-point number 100.0 and is not allowed in the address expressions).

As you type, the list at the bottom will show you all labels that contain the typed text as a substring. The label we are searching for is *_Break* in the module *Test*, therefore its full name is *Test._Break*. Click it once and press **Follow label**. (For some subtle reasons, doubleclick does not work):



Disassembler repositions to the procedure *_Break*. Note that *Test.exe* is under permanent development, and addresses may differ from the picture. The procedure consists of two commands: NOP, followed by RETN, and does perfectly nothing. Label *_Break* points to the command NOP with hexadecimal code 0x90. This is what the message in the debugging application means.

Line with NOP is still selected? Now press **F2**. This is a shortcut for "Toggle breakpoint". Red address background indicates that unconditional software breakpoint was set. OllyDbg has replaced the byte at address 004023AB with INT3:

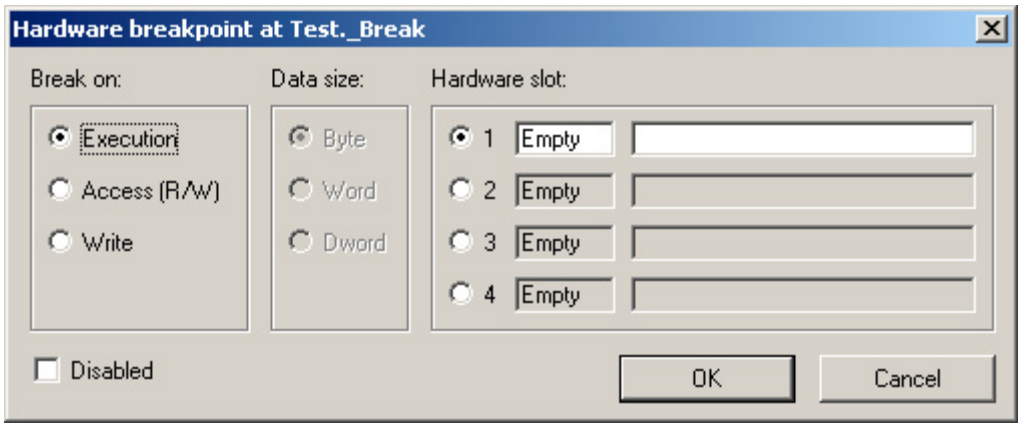


Press "**Read [_Break]**" again. Now the message is different: "**[BYTE _Break] = 0xCC (INT3)**". So the memory of the process is pnysically changed. But Disassembler still displays the old command? Yes, because it takes care of the breakpoints and replaces them with original data before the memory is displayed. But, as you can see, soft breakpoints are visible to the application. For example, some viruses check critical system functions and don't do any evil if there are breakpoints set. Moreover, if you set soft breakpoint on data, the program may misbehave or crash. **You are allowed to set soft breakpoints only on the first byte of the command!**

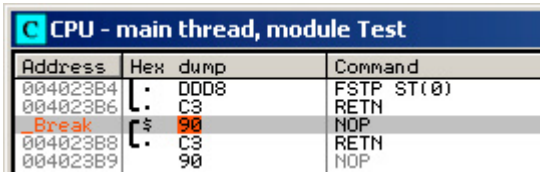
The breakpoint is set and application is active. Press "**Call _Break**". OllyDbg pops up. The status of the debugged program is changed to "Paused" and there is a message, "**Breakpoint at Test._Break**". What happened? Command INT3 has generated an interrupt. Windows has paused the application and passed this interrupt to the debugger. Now you can do whatever you want - read registers, analyse memory or make patches.

Press **F9** or button  (Run). Program continues execution. Assure that line with breakpoint is still selected and press **F2** again to remove the breakpoint.

Now right click tyhe command and select **Breakpoint | Hardware...** from the menu. OllyDbg will ask you when this breakpoint should trigger: when command executes, when memory is accessed or when memory context changes. Because this memory location is recognized as a code, OllyDbg suggests breakpoint on execution:

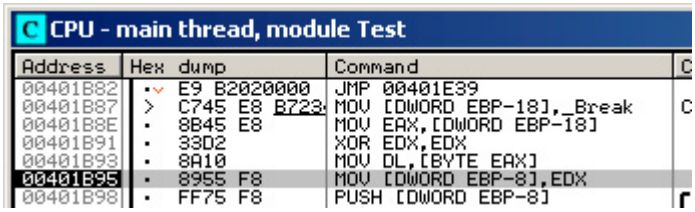


Accept this suggestion by pressing **OK**. Now red background appears in the memory dump, indicating hardware or memory breakpoint:



Press "**Read [_Break]**". The code is NOP, hardware breakpoints are invisible to the application. (Well, not exactly. Application may read thread's context and analyse debugging registers). Now press "**Call _Break**" and the breakpoint will be hit.

Run program, call hardware breakpoint dialog again and change breakpoint type to **Access (R/W)**. Press "**Call _Break**" - nothing happens. Press "**Read [_Break]**". The application will stop, but this time not at address *_Break*:



Note the message in the status bar:

Hardware breakpoint on data access triggers *after* data is accessed. Check register EAX. It contains 004023B7 - in our example, this is the addres of the procedure labelled as *_Break*. This address was read in the command at 00401B93: MOV DL,[BYTE EAX]. But hardware breakpoint was reported at the following command.

Hardware breakpoints are very convenient. If, by mistake, you set hardware breakpoint on data, nothing will happen. Hardware breakpoint on execution will not trigger if memory location is accessed as a data. But hardware breakpoints are very scarce. There are only 4 of them, and there are some limitations concerning their alignment. Therefore it's time to learn memory breakpoints.

Remove the hardware breakpoint. Oh, Disassembler points to different memory location and you don't know how to come back? Either call Go to window, or click EAX in Registers pane and from the pop-up menu choose **Follow in Disassembler**. You may also call Disassembler menu and choose **Go to | Previous location** or press gray Minus button on the digital block (rightmost on your keyboard, bad luck if your laptop is not full-size). Disassembler and Dump keep separate histories of displayed addresses and allow you to walk these histories in both directions.

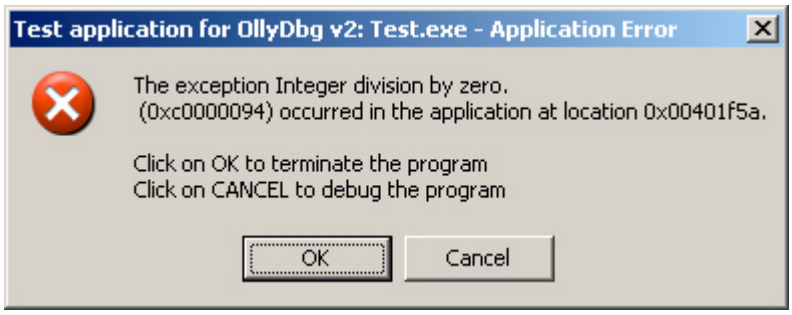
After hardware breakpoint is removed, select **Breakpoint | Memory...** and check **On execution**. Run the program. Some strange thing happens - status bar starts blinking. It displays something like "**6000 events per second**". Memory breakpoints use hardware memory protection of the 80x86 architecture. It allows to disable access to individiual memory pages. Each memory page is 4096 bytes large. If one disables memory access on execution, then each time any command from this memory block is hit, hardware generates an exception. If exception is a false positive, OllyDbg continues execution but counts this event in the status bar. Exception processing takes significant time. **Memory breakpoints may be very slow**. But the number of memory breakpoints is unlimited.

Press "**Read [_Break]**". As expected, memory breakpoint is not visible to the application. Press "**Call _Break**" and OllyDbg will report memory breakpoint on execution.

OllyDbg preserves breakpoints between debugging sessions in the .udd file. To verify this, close OllyDbg and restart it again. Reload Test.exe (**Ctrl+F2** or  - "rewind"), run it and press "**Call _Break**". The program will pause at the same position as the last time.

Lesson 2 - Patching the code

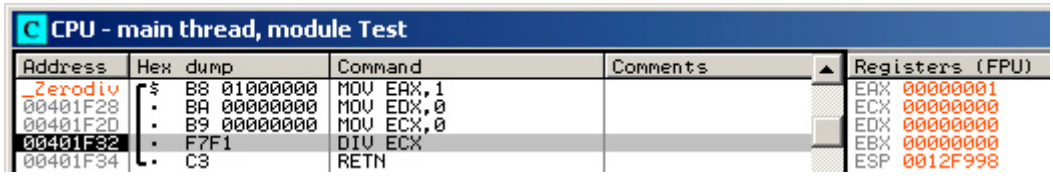
Start *Test.exe* (without OllyDbg!) and press button labelled "**0 : 0**". Application will call routine that divides integer 1 by integer 0. As expected, program crashes:



We are going to locate invalid command and remove it from the code.

Load *Test.exe* into the OllyDbg and run it (**F9**). Now press button "**0 : 0**" again. The execution will pause and status bar at the bottom will display the message "**Integer division by zero - Shift+Run/Step to pass exception to the program**".

Look at the CPU Disassembler pane:



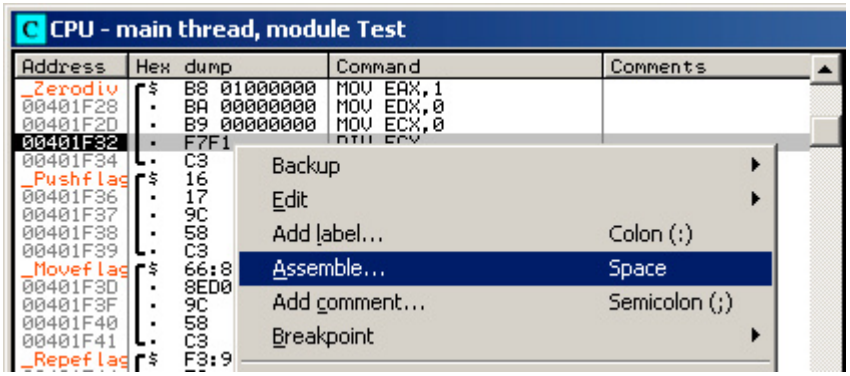
EIP points to the command DIV ECX. If you don't know what this command does, press **Shift+F1** (or select **Help on command** from the context menu):

Brief glance into Registers - ECX is currently 00000000, hence division by zero. We are in the subroutine labelled as *_Zerodiv* with the following code:

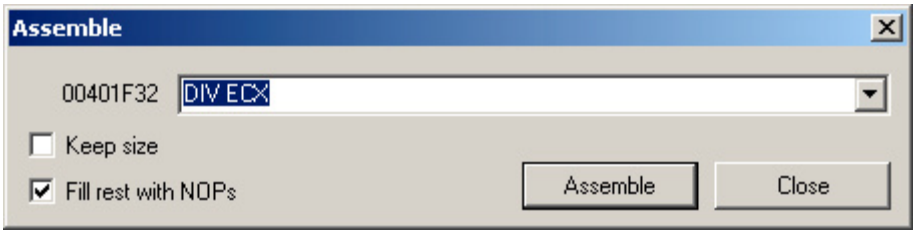
```
MOV EAX,00000001
MOV EDX,00000000
MOV ECX,00000000
DIV ECX
RET
```

Now we have several options. We may comment the whole routine out, replacing MOV EAX,00000001 with RET. Or we may load ECX with 1 instead of 0. Or we may replace DIV ECX with NOPs. Let's choose the third method.

Click on DIV ECX. When this command is selected, right click on it and choose **Assemble...** from the menu:



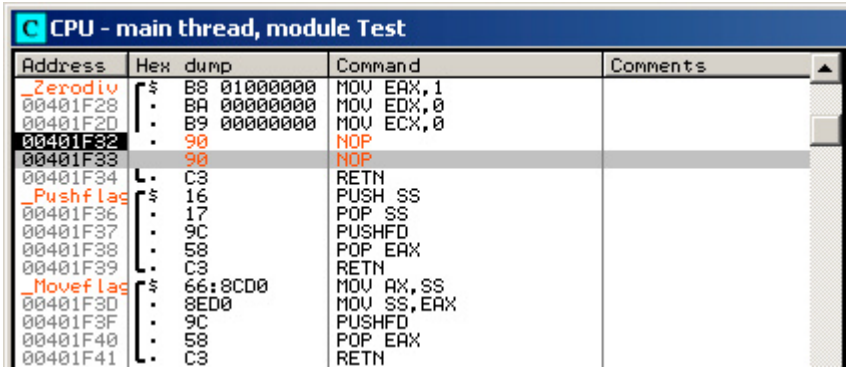
Assemble window will appear:



Note two checkboxes. If **Keep size** is checked, you will be able to modify only the selected memory (in our case, as DIV ECX is 2 bytes long, only the addresses 00401F32 and 00401F33). This prevents from unwanted overwriting of important code.

New command is not necessarily as long as the original. Suppose you want to remove short jump to the destination that is 8 bytes back, binary code EB F4. You replace it with NOP (90). But NOP is only one byte long. What you will get is the sequence 90 F4 that disassembles to NOP; HLT. If you attempt to execute this sequence, the program will crash, reporting privileged instruction. Uncontrolled command remnants can be dangerous. Therefore I recommend to always activate **Fill rest with NOPs**. If new command does not fit exactly, the rest will be automatically filled with harmless NOPs.

Back to the lesson. Check **Fill rest with NOPs,** type NOP and press **Assemble** (or **Enter** key on the keyboard). Window will ask you to enter next command, but we don't need it. Close dialog. This is what you will see:

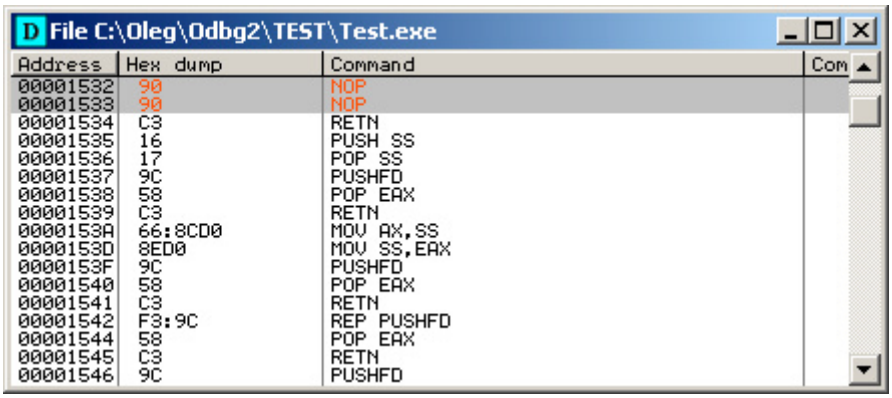


Command DIV ECX was replaced by a sequence of two NOPs. Note that modified commands are highlighted. To keep track of modified commands, OllyDbg has created backup of the old code section. This operation is memory-consuming (backup is as large as the code section) but very useful. You can

look at the old code, undo your changes or search for modifications. (Backup is also helpful if code you are debugging is self-modifiable).

Also note that parenthesis which indicates the extent of the procedure *_Zerodiv* is now broken. OllyDbg removes analysis data from the modified code.

Run program and press "**0 : 0**" again. Nothing happens. This indicates that our patch is correct. But changes will be lost when application terminates. We need to copy them back to the executable file. Usually this is a complicated task, but OllyDbg takes care of it. Select modified code (two NOP commands), call context menu and choose **Edit | Copy to executable**. OllyDbg will create new dump window, read executable file, locate address corresponding to the selection and copy our modifications to the file dump:



You may add other patches, either through CPU Disassembler or directly. After you are ready, right click in the file dump and choose **Save file...** You will be asked to confirm this action, just press **Yes**. Another dialog will appear, asking you to select file name. We don't want to change the original *Test.exe*. (This may be not possible if test application is still running). Select different name, say, *Test1.exe* and press **Save**.

Start *Test1.exe* and press "**0 : 0**". There is no exception. We have successfully patched the program.

Lesson 3 - Run trace

When program executes jump to wrong location, it may be extremely hard to find the location of the invalid jump.

There is a feature called Run trace. When you start run trace, OllyDbg executes debugged application step by step, one command at a time, and protocols execution.

Run trace is slow, slooooow. Modern CPU can execute several billions commands in a second. When run trace stops debuggee after each command, execution speed is limited to maybe 30 thousand commands. If fast command emulation is enabled (**Options | Debugging | Allow fast command emulation**), OllyDbg usually traces 300,000 to 600,000 commands per second. This speed is sufficient for simple GUI applications like *Test.exe*.

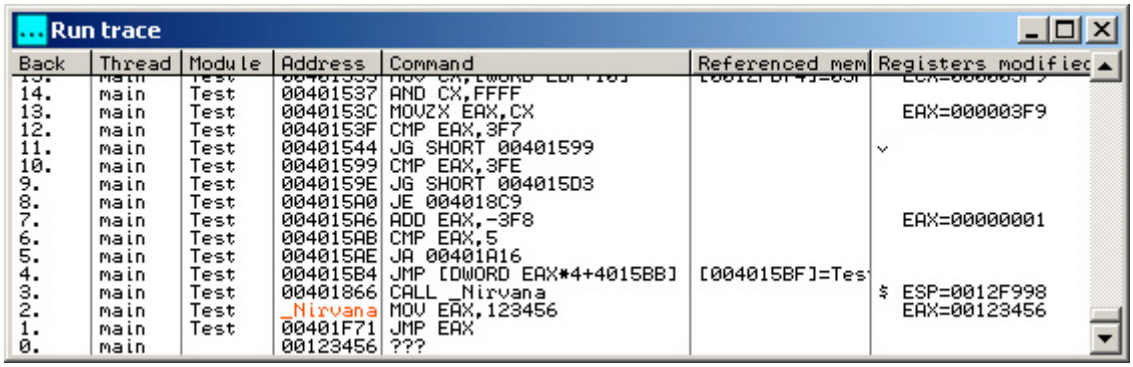
Another drawback of run trace is that currently it can trace only single thread. If error is caused by the interaction of two threads, run trace will never catch it. But *Test.exe* has only one thread.

Now let's begin our lesson. Open *Test.exe* in OllyDbg, start it and press button labelled "**JMP 123456**". Exception! OllyDbg reports "Access violation when reading [00123456]", but CPU Disassembler is empty. There is no memory at address 00123456.

Next attempt. Set run trace options (**Options | Run trace**). In our case, the only important option is **Don't**

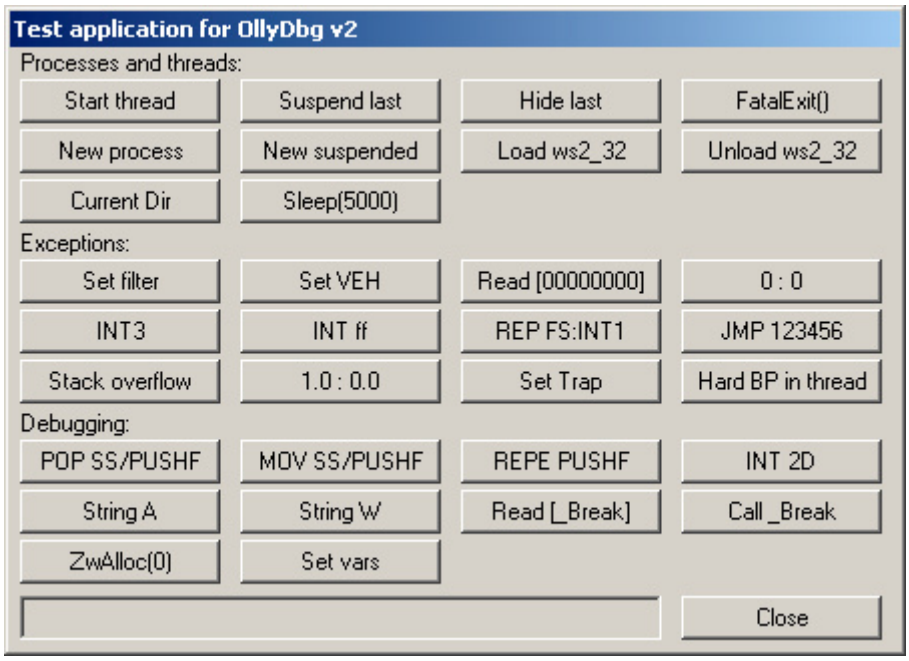
enter system DLLs. It must be *unchecked* because jump is executed from the window's callback function on button message. Restart the application (**Debug | Restart** or press) and run it. Changing debugging mode "on the fly" is not allowed. Pause program ( ) and start run trace ( ). Note that status changes to "Tracing" and status bar blinks displaying something like "120672 events per second". Run trace causes many debugging events, usually one per executed command. Switch to the application's window (note how slowly it redraws) and press "**JMP 123456**".

Instead of crash, this time you will see the error "OllyDbg is unable to step over the command at (possibly invalid) address 00123456. Memory is not readable". Cancel it and open run trace protocol (**View | Run trace** or):



The last command (or, rather, the lack of such) is placed at address 00123456. The last but one is JMP EAX at address 00401F71. As one can see from the last column, register EAX at this moment contains 00123456. Bingo! We have found the source of the problem.

Test.exe



Test.exe allows you to learn extended features of the OllyDbg. We have already used it in the lessons. Most of the code examples in this manual are based on this application.

You may download the source code of *Test.exe* from my site, www.ollydbg.de. Note that I constantly add new features and may modify existing. Therefore addresses may differ and new buttons may be added. To find code, use exported labels.

Here is the brief explanation of actions associated with each button. Italic text in the parenteses is the name of the associated label, if any:

Start thread (*_Thread*)- starts new thread that opens window and counts (rather imprecisely) 100-ms intervals. If you close window, thread terminates;

Suspend last - suspends last created thread. You can resume it from the OllyDbg's Threads window;

Hide last - attempts to hide last created thread from the debugger (*NtSetInformationThread()*, code 0x11). This feature is OS-dependent. If hiding is not supported, you will see the error message;

FatalExit() - calls *FatalExit(0)*;

New process - starts new instance of *Test.exe*. If option **Debug child processes** is checked, OllyDbg will start new instance and attach it to the newly created process;

New suspended - same as above, but new process is initially in the suspended state (does not run);

Load ws2_32 - loads library *ws2_32.dll*. Why ws2_32? Because it is available on all Windows versions and loads another library, *ws2help.dll*;

Unload ws2_32 - asks to unload *ws2_32.dll* from the memory;

Current Dir - displays current directory as reported by *GetCurrentDirectory()*;

Sleep(5000) - executes *Sleep(5000)* - 5 seconds long pause;

Set filter - installs custom filter for unhandled exceptions by calling *SetUnhandledExceptionFilter()*. Note that debugger is normally not allowed to pass exception to this filter. OllyDbg uses tricks to persuade OS to call this filter. This behaviour is controlled by the option **Pass unprocessed exceptions to Unhandles Exception Filter**;

Set VEH - adds handler to the chain of the Vectored Exception Handlers (VEH). To protect application from viruses, Windows scrambles this chain using session-unique key. Therefore OllyDbg is able to walk VEH only if process was created by the OllyDbg itself and not if it was attached to the already running process;

Read [00000000] (*_Accessviolation*) - asks *Test.exe* to read contents of the memory at address 00000000. Usually this leads to memory access violation. However, you may press **ZwAlloc(0)** to allocate memory block at location 0!

0:0 (*_Zerodiv*) - divides 0 by 0, causing integer division by zero;

INT3 (*_Int3*) - executes INT3;

INT ff (*_Intff*)- executes INT 0FFh;

JMP 123456 (*_Nirvana*) - executes jump to unallocated memory page JMP 0123456h;

Stack overflow - makes infinite recursive call, overflowing stack;

1.0:0.0 (*_Fzerodiv*) - divides floating point number 1.0 by floating 0.0, causing floating-point division by zero. Note that due to the logic of floating coprocessor, this error is reported on the next FPU command. To find invalid command, one must follow **Last cmnd** in the Registers pane;

Set Trap (*_Settrap*) - sets bit T (Single-Step Trap) in the register EFL, causing Single Step exception;

Hard BP in thread - sets hardware debugging breakpoint 0 in the last created thread, causing Exception Single Step (all debugging exceptions are routed to the same interrupt and Windows makes no attempts to distinguish between them);

POP SS/PUSHF (*_Pushflags*) - executes commands PUSH SS; POP SS; PUSHFD and displays contents of the top of the stack. When OllyDbg 1.10 traced this sequence, it pushed flag T;

MOV SS/PUSHF (*_Moveflags*) - executes commands MOV AX,SS; MOV SS,AX; PUSHFD with effect similar to the previous button;

REPE PUSHF (*_Repeflags*) - executes undocumented command REPE:PUSHFD. When OllyDbg 1.10 traced this command, it pushed flag T;

INT 2D (*_Int2d*) - executes INT 02Dh;

String A - executes **OutputDebugStringA("Debug string (ASCII)")**;

String W - executes **OutputDebugStringW(L"Debug string (UNICODE)")**;

Read [_Break] - reads and displays byte at address *_Break*. This address is an entry of a small subroutine that never executes:

```
_Break:  NOP
         RET
```

Normally, if you press the button, test routine repotrts **[BYTE Break] = 0x90 (NOP)**. Try to set soft breakpoint at address *_Break* and press button again. Now the message reads: **[BYTE Break] = 0xCC (INT3)**, indicating that NOP is replaced by a 1-byte breakpoint INT3.

Call _Break - calls function *_Break()* described above. It does nothing, unless breakpoint is set.

ZwAlloc(0) - allocates 1 page (4096 bytes) of memory at base address 0x00000000. Yes, this is possible!

Set vars (*_ppi* and *_rect* in data section) - increments internal counter by one and assigns its value to variables ***ppi* and *rect[2][3].right*. These variables are declared as

```
int **ppi;
RECT rect[4][4];
```

There are also several places that highlight the features of analysis:

Nestetdloops - five short nested loops:

```
int _export Nestedloops(void) {
  int i,j,k,l,m,n;
  n=0;
  for (i=0; i<10; i++) {
    for (j=0; j<20; j+=2) {
      for (k=0; k<30; k+=3) {
        for (l=0; l<40; l+=4) {
          for (m=0; m<50; m+=5) {
            n++;
          };
        };
      };
    };
  };
  return n;
};
```

Nestedcalls - calls to functions that get calls to functions as their arguments:

```
int _export Sum(int x,int y) {
  return x+y;
};

int _export Nestedcalls(void) {
  int n;
  n=Sum(Sum(Sum(1,2),Sum(3,4)),Sum(Sum(5,6),Sum(7,8)));
  n+=MulDiv(MulDiv(1,1,1),MulDiv(1,1,1),MulDiv(1,1,1));
  return n;
};
```

Assembler and disassembler

General information

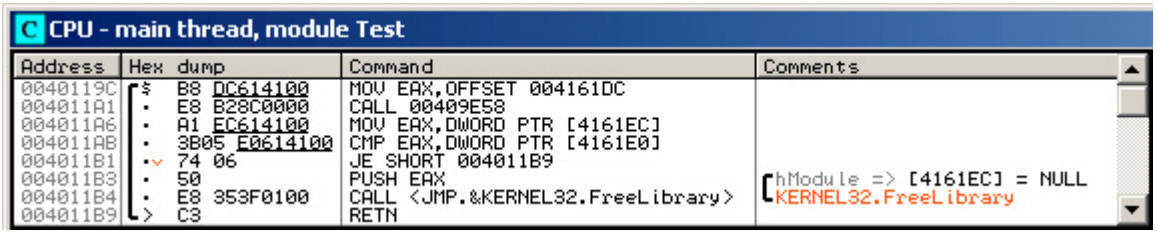
OllyDbg 2.01 supports all 80x86 commands, FPU, MMX, 3DNow!, SSE, SSE2, SSE3, SSSE3, SSE4 and AVX extensions. Please note following peculiarities and deviations from Intel's standard:

- REP RET (AMD branch prediction bugfix) is supported;
- Multibyte NOPs (like NOP [EAX]) are supported. However, Assembler always attempts to select the shortest possible form, therefore it may be hard to set the required NOP length;
- FWAIT is always separated from the following FPU command. Assembler never adds FWAIT, for example, FINIT is in fact translated to FNINIT etc.;
- Assembler understands no-operand forms of binary FPU commands (like FADDP), but Disassembler always uses two-operand form (FADDP ST(1),ST);
- LFENCE: only form with E8 is accepted (0F AE E8);
- MFENCE: only form with F0 is accepted (0F AE F0);
- SFENCE: only form with F8 is accepted (0F AE F8);
- PINSRW: register is decoded as 16-bit (only low 16 bits are used anyway);
- PEXTRW: memory operand is not allowed, according to Intel;
- INSERTPS: source XMM register is commented only as a low float, whereas command may use any float;
- Some FPU, MMX and SSE commands accept either register only or memory only in ModRM byte. If counterpart is not defined, Disassembler reports it as an unknown command. Integer commands, like LES, report in this case invalid operands.
- SSE4 commands that use register XMM0 as a third operand are available both in 2- and in 3-operand formats, but Disassembler will show only the full 3-operand form;
- Assembler accepts CBW, CWD, CDQ, CWDE with explicit AL/AX/EAX as operand. Disassembler shows only implicit no-operand form;
- 16-bit AVX floating-point numbers are decoded as hexadecimal short numbers.

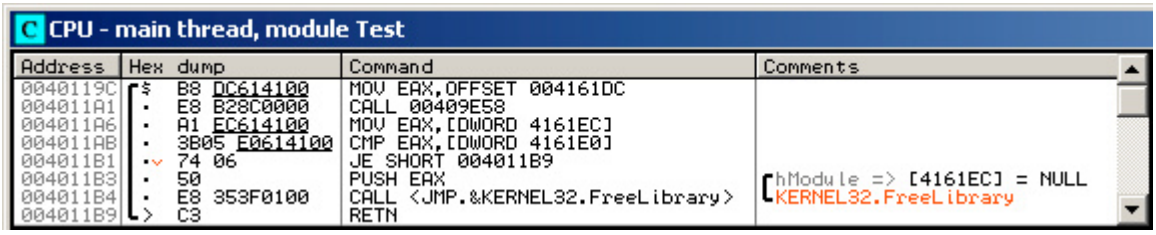
Disassembling modes

OllyDbg supports four different decoding modes: MASM, Ideal, HLA and AT&T. They are controlled by the option **Code | Disassembling syntax**.

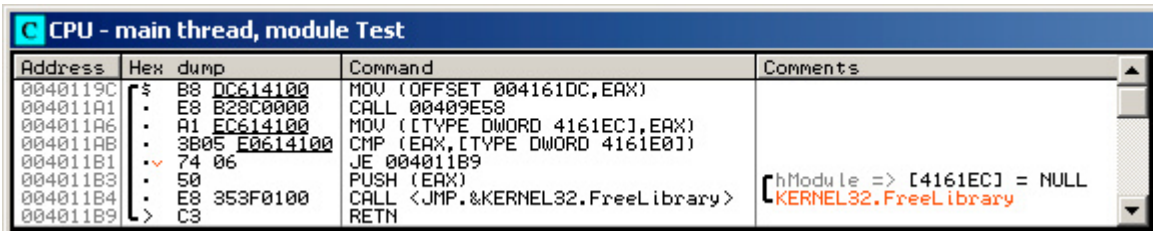
MASM is the de facto standard of the 80x86 assembler programming:



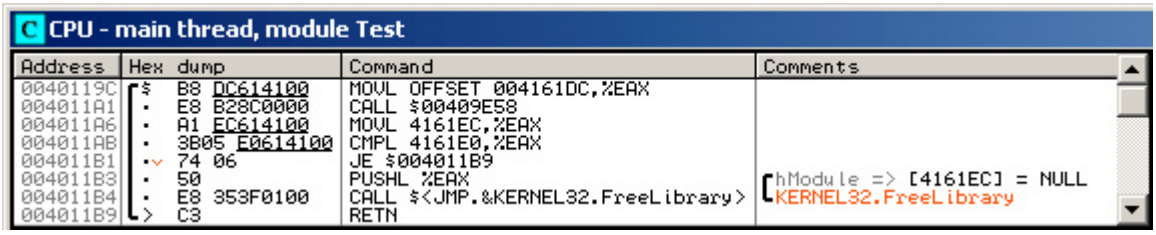
Ideal mode, introduced by Borland, is very similar to MASM but decodes memory addresses differently:



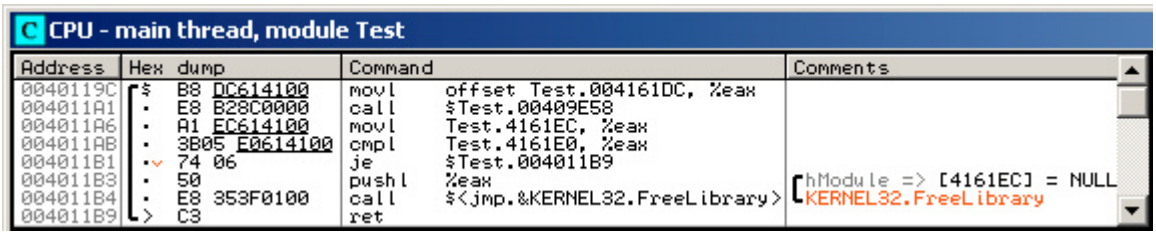
High Level Assembly language, created by Randall Hyde, uses yet another, functional syntax where first operand is a source and operands are taken into the brackets. HLA is a public domain software, you can download it together with excellent documentation, tutorials and sources from http://webster.cs.ucr.edu. Example of HLA syntax:



AT&T syntax is popular among the Linux programmers:



Disassembler is configurable. Here is another possible layout in the AT&T mode:



Demangling of symbolic names

In C++ you are allowed to declare functions with the same name but with the different number or types of parameters. Linker must be able to distinguish them one from another. To assure this, compiler adds the description of the formal parameters and type of return to the function's name. This process is called name decoration, or else name mangling. For example, *void Setdlgscroll(HWND hparent,int id,int pos)* will be encoded as *?Setdlgscroll@@YAXPAUHWND__@@HH@Z* in Visual Studio and as *@Setdlgscroll$qp6HWND__ii* in Borland C++ Builder. Functions *Floatingcall()* and *Floatingargs()* in Test.exe have mangled names *@Floatingargs$qfdg* and *@Floatingcall$qv*.

Restoring of the original names is called demangling. OllyDbg can demangle names created by GNU, Microsoft and Borland compilers. This process is controlled by the option **Code | Demangle symbolic names**. Unlike the OllyDbg v1.xx, version 2 keeps both versions and can reswitch between the two

presentations anytime.

Although it's possible to extract the number and types of the parameters, OllyDbg restores only the name of the function. If demangling is on, it may happen that several memory locations will be named identically.

Conditional commands

Look at these two command sequences:

```
83F8 00   CMP EAX,0          09C0     OR EAX,EAX
74 50     JE 00123456        74 51    JE 00123456
```

They do the same: jump is taken if EAX is zero. In the first case mnemonics JE perfectly describes the action: jump is taken if EAX and 0 are equal. But what JE means in the second example? That EAX and EAX are equal? Of course not. Jump is taken if *result* of the operation (EAX OR EAX) is zero. In this case, mnemonics JZ would be better. The problem is that JE and JZ are synonims and translate to the same binary code 74 (or 0F84 if offset exceeds 128 bytes). How can OllyDbg guess which one is better?

The answer: analysis. When OllyDbg analyses code, it notices which command has set conditional flags and uses this information to select mnemonics. The feature is controlled by the option **Mnemonics | Guess alternative forms of conditional commands**. and influences decoding of JE/JZ, JNE/JNZ, JAE/JC and JB/JNC (and correspondingly SETE/SETZ, CMOVE/CMOVZ etc.)

Assembler syntax

Assembler automatically recognizes MASM, Ideal and HLA syntax. AT&T is not supported.

Dependless on the selected disassembling mode, you may type commands in MASM, Ideal or HLA format. HLA is recognized by the parentheses around the operands. The following commands assemble identically:

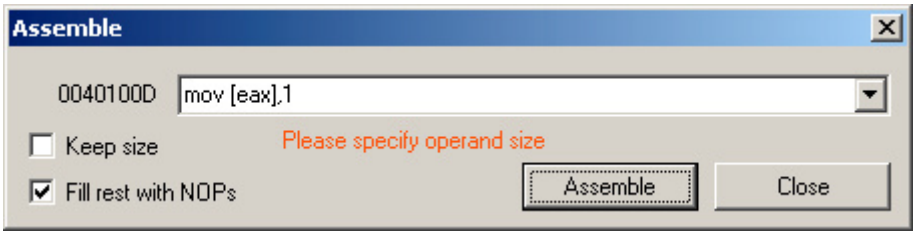
```
MOV [EAX],ESI
MOV [DWORD EAX],ESI
MOV DWORD PTR [EAX],ESI
MOV (ESI,[EAX])
```

(Note inverse order of operands in HLA). Memory operands require square brackets: MOV EAX,_Zerodiv moves address of label _Zerodiv into register EAX, MOV EAX,[_Zerodiv] moves contents of doubleword memory at address _Zerodiv into EAX, and MOV _Zerodiv,EAX is invalid.

You don't need to specify the size of memory operand if it can be derived from the command or from other operands. For example, Assembler knows that register ESI has doubleword size and therefore memory operand in MOV [EAX],ESI is also a 32-bit location. But in the case of MOV [EAX],1 all three possible forms are equally valid:

```
MOV [DWORD EAX],1
MOV [WORD EAX],1
MOV [BYTE EAX],1
```

and Assembler will issue a warning:



You may specify known constants directly as command operands:

MOV EAX,WM_PAINT	translates to	MOV EAX,0F

Expressions in operands are limited to addition, subtraction and ORing of constants and labels. All three operations have the same priority and are executed from left to right. 32-bit overflows are ignored. Parentheses are not allowed:

MOV EAX,WS_CHILD|WS_VISIBLE	translates to	MOV EAX,50000000

All constants in the OllyDbg are hexadecimal by default. If you want to specify decimal constant, follow it with the point:

MOV EAX,1000	is equivalent to	MOV EAX,0x1000
MOV EAX,1000.	translates to	MOV EAX,3E8

Hexadecimal constant may begin with a letter (A-F), but symbolic labels have higher priority than hex numbers. Assume that you have defined label DEF at address 0x00401017. In this case,

MOV EAX,ABC	translates to	MOV EAX,0ABC
MOV EAX,DEF	translates to	MOV EAX,401017

To avoid ambiguity, precede hexadecimal constants with 0 or 0x: MOV EAX,0DEF .

There are only few exceptions to this rule. Indices of arguments and local variables are decimal. For example, ARG.10 is the address of the tenth call argument with offset $10_{10} \cdot 4 = 40_{10} = 0x28$. To memorize this rule, note that ARG and index are separated with a decimal point.

Ordinals are also in decimal. COMCTL32.#332 means export with ordinal 332_{10}.

16-bit addresses are supported, but Assembler always assumes 32-bit code segment and adds address size prefix 0x67. If address contains no 16-bit registers, use keyword SMALL to force 16-bit addressing mode:

MOV ECX,[DWORD FS:0]	generates 32-bit address
MOV ECX,[SMALL DWORD FS:0]	generates 16-bit address (command is 1 byte shorter)

Undocumented 80x86 commands

OllyDbg recognizes several undocumented 80x86 commands:

Command	Hex code	Comments
INT1 (ICEBP)	F1	1-byte breakpoint

SAL	D0 /6, D2 /6, C0 /6	Arithmetic shift, identical with D0 /4 etc.
SALC	D6	Set AL on Carry Flag
TEST	F6 /1	Logical Test, identical with F6 /0
REPNE LODS, REPNE MOVS, ...	F2:AD, F2:A5, ...	String operations, REPNE is interpreted the same way as REP
FFREEP	DF /0	Free Floating-Point Register
UD1	0F B9	Intentionally undefined instruction

Disassembler supports all mentioned commands. Assembler will not generate non-standard SAL and TEST commands; if necessary, use binary edit to create binary codes.

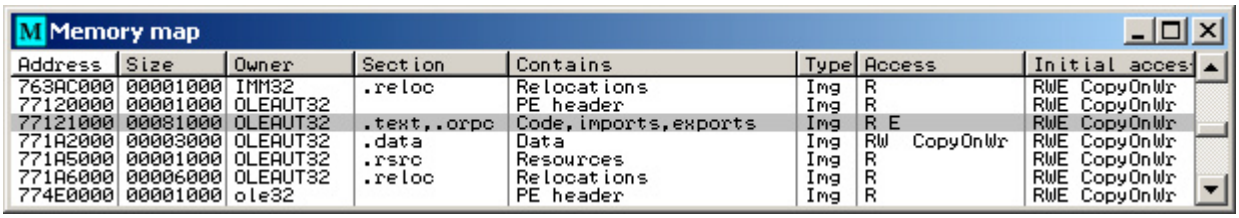
Memory map

General information

Each 32-bit application runs in its own virtual 2^{32}-byte memory space. Only the lower part of this space (2 or 3 gigabytes) is available to the application. Windows fills it with executable modules, data blocks, stacks and system tables. Minimal allocation unit is a page (4096 bytes). Each page has several attributes that indicate whether page can be read, modified or executed. Except for this protection, there are no physical borders berween the memory blocks. If application allocates two blocks of data and they are adjacent by accident, nothing will prevent appplication from treating them as a single data block.

OllyDbg treats application's memory as a set of independent blocks. Memory map window displays all memory blocks available to the Debuggee. There are no standard means to determine where one block ends and another begins, so it may happen that OllyDbg will show several portions of allocated memory as a single memory block. But in most cases precise resolution is not necessary.

Any operation on the memory contents is limited to the block. In most cases, this works fine and facilitates debugging. But if, for example, module contains several executable sections, you would be unable to see the whole code at once. Therefore OllyDbg may merge adjacent code block into one, as on the following example:



System DLL *oleaut32.dll* in WindowsXP declares two adjacent sections: *.text* and *.orpc* as code. They have identical attributes, and OllyDbg interpretes these two sections as a single memory block.

Memory map is updated on every pause. If you need to actualize the window while program is running, choose **Update** from the pop-up menu or press **Ctrl+R** (as in Re-read).

Doubleclick on the line in Memory map opens standalone window that displays the contents of the pointed memory block.

You may also search the whole memory of the process for the specified combination of bytes, starting from the selected block (**Search...** or **Ctrl+B**, see detailed description below in the chapter Search). When the combination is found, OllyDbg opens dump window and scrolls it to the first found location. Press **Ctrl+L** to find next location in the dump or **Esc** to close dump. Press **Ctrl+L** in the Memory map to continue search.

Kernel memory

Kernel (system) memory is allocated in the high memory area. Usually it starts at address 0x80000000 (home versions) or 0xC0000000 (some server versions of Windows).

If OllyDbg can see kernel memory, it will attempt to display its contents using special debugging functions. These functions work well under Windows XP, but usually fail under Windows 7. OllyDbg displays kernel memory as a single block where unavailable pages are filled with zeros instead of question marks, as in all other cases. OllyDbg can neither modify kernel memory, nor change memory attributes, nor set breakpoints of any kind. The only available operation is search. Attention, full search may take several minutes!

Kernel memory is not directly accessible to the applications, but OS or driver may grant access to the portions of this memory. For example, some antivirus programs redirect *LoadLibrary*, *GetProcAddress* and similar API functions to the kernel memory, where antivirus may make additional validity checks without the risk of being corrupted by the virus. If OllyDbg encounters such function, it traces it using single-step traps.

Backup

For each memory block, except for kernel, you can create a backup. This is a copy of the current memory contents. If backup is available, OllyDbg highlights differences. You can view backup, restore modifications, save backup to disk or read it back. Saved backup is helpful if you want to see the differences between two runs.

Backup is created automatically when you edit code or data in the CPU window. Backup is necessary if you request pause on modified command while running run trace. See detailed description below in the section "Run trace and profiling". Option **Debugging | Auto backup user code** tells OllyDbg to create backups of executable code at the moment when module is loaded into the memory.

Note that standalone dump windows may create their own backups. They are physically different from the backups described here.

Break on memory access

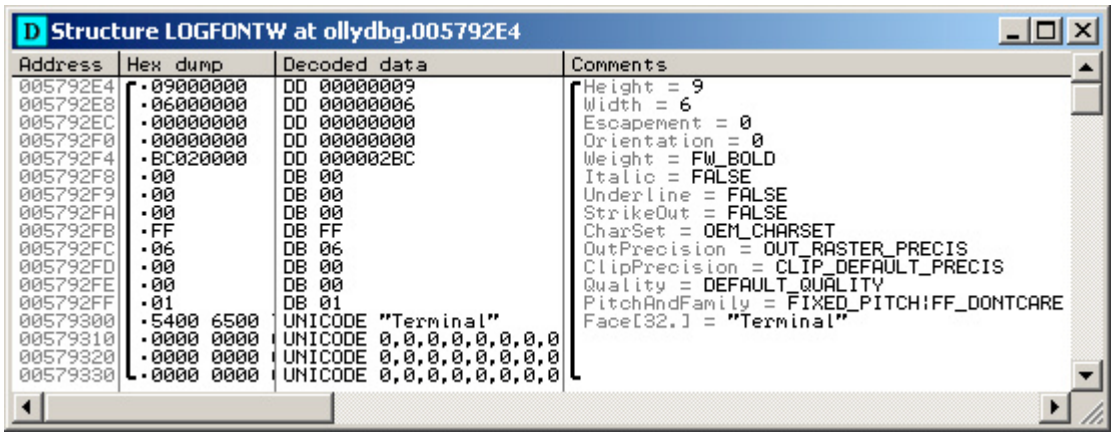
In the Memory map window you can set one-shot breakpoint on access on the whole memory block. Any possible access (read, write or execution) will trigger a break and remove breakpoint. Break on stack is not allowed, it may lead to crash inside system calls.

This kind of memory breakpoint is useful if you need to find calls to DLL or returns from such calls. Just place breakpoint on the code section of the corresponding module.

Dumps

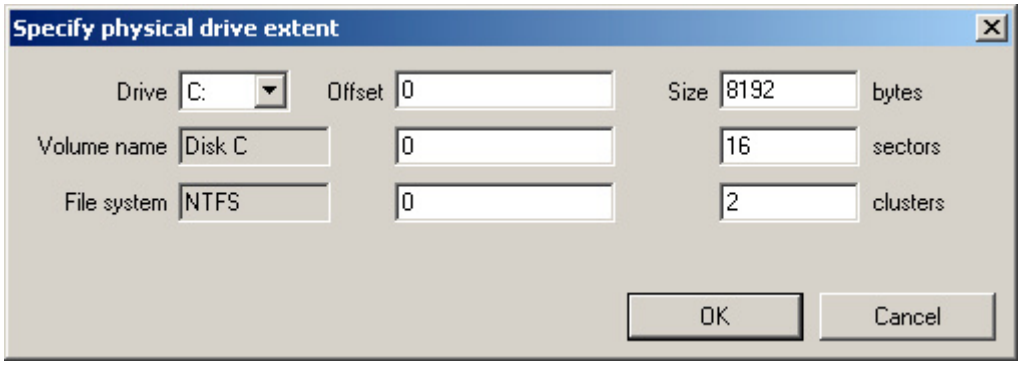
Dump windows display contents of memory, file or disc. CPU window includes three dumps: Disassembler, Dump and Stack. One may also open unlimited number of standalone dump windows.

There are several ways to create a dump. Doubleclick on the line in the Memory map to open a dump of the selected memory block. Select a piece of memory in CPU Disassembler or CPU Dump and choose **Open in a separate Dump window** from menu to monitor only this piece. If you choose **Decode as structure...**, the selection can be decoded as one of the predefined Windows structures. If you choose **Decode as pointer to structure...**, doubleword that starts at the first byte of selection will be interpreted as a pointer to the structure, like *LOGFONTW* passed to *CreateFontIndirectW()* on the following example:



To open a file, select **View | File...** from the main menu. If file extension differs from *.exe*, *.dll* and *.lnk*, you can drag and drop it into the OllyDbg. To save modified file, choose **Save file...** from the pop-up menu.

If you have sufficient administrative rights, you may directly inspect the contents of the local disks on your computer (**View | Drive...**):



The size of the extent is limited mainly by the available memory. Drive dump is read-only, you can't save any modifications back to the disk - it would be too dangerous for the file system.

Dumps support many different formats: hexadecimal with ASCII, UNICODE or MBCS text, pure ASCII or UNICODE text, 16-and 32-bit integers in all possible presentations, stack view, 32-, 64- and 80-bit floating-point numbers, or as a disassembled code. For ASCII and MBCS you can choose one of the codepages installed on your computer. Here is an example of the multibyte (variable-width) UTF-8 dump:

Address	Hex dump	Multibyte (UTF-8)
00418A3B	4F 6C 6C 79 44 62 67 E4 B8 AD E6 96 87 E7 AB 99	O l l y D b g 中文站
00418A4B	E6 98 AF E6 9C 80 E6 9D 83 E5 A8 81 E7 9A 84 4F	是最权威的O
00418A5B	6C 6C 79 44 62 67 E4 B8 AD E6 96 87 E8 B5 84	l l y D b g 中文资
00418A6A	E6 96 99 E7 AB 99 E3 80 82 4F 6C 6C 79 44 62 67	料站。O l l y D b g
00418A7A	E6 98 AF E4 B8 80 E7 A7 8D E5 85 B7 E6 9C 89	是一种具有
00418A89	E5 8F AF E8 A7 86 E5 8C 96 E7 95 8C E9 9D A2	可视化界面
00418A98	E7 9A 84 33 32 E4 BD 8D E6 B1 87 E7 BC 96	的3 2 位汇编
00418AA6	E5 88 86 E6 9E 90 E8 B0 83 E8 AF 95 E5 99 A8	分析调试器
00418AB5	EF BC 8C E6 98 AF E4 B8 80 E4 B8 AA E6 96 B0	，是一个新
00418AC4	E7 9A 84 E5 8A A8 E6 80 81 E8 BF BD E8 B8 AA	的动态追踪
00418AD3	E5 B7 A5 E5 85 B7 EF BC 8C E5 B0 86 20 49 44 41	工具，将 I D A
00418AE3	E4 B8 8E 53 6F 66 74 49 43 45 E7 BB 93 E5 90 88	与S o f t I C E 结合
00418AF3	E8 B5 B7 E6 9D A5 E7 9A 84 E6 80 9D E6 83 B3	起来的思想
00418B02	EF BC 8C 52 69 6E 67 33 E7 BA A7 E8 B0 83	，R i n g 3 级调

"Variable-width" means that single character may be represented by one, two or more bytes and that the length of the encoding depends on the character. OllyDbg supports at most five bytes per character, which is sufficient for all currently existing sets. Note that character 'O' at address 00418A3B above is encoded as a single byte 4F, whereas the selected Chinese Han character that starts at 00418A42 requires three bytes: E4, B8, AD.

First displayed dump line contains 10 UTF-8 characters: seven Latine ("OllyDbg") and three Han. Multibyte dump has 16 positions. 6 unused positions at the end of the line are filled with the grayed ellipsises to distinguish them from the significant spaces.

OllyDbg never splits hexadecimal presentation of recognized multibyte characters between the dump lines. Therefore some dump lines contain less than 16 bytes.

Han, Kanji or Hangul symbols are significantly wider that ASCII and would be clipped if one attempts to force them into the ASCII raster. To avoid clipping, activate option **Dump | Use wide characters in UNICODE & multibyte dumps**. ASCII and multibyte code page can be changed any time from the **Dump** pane in options or directly from the pop-up menu.

Arrow buttons on the keyboard move selection one item at a time (byte in the hex dumps, wide character in the UNICODE, command in Disassembler). Press Shift and move selection to select multiple items.

Key combinations **Ctrl+UpArrow** and **Ctrl+DownArrow** scroll dump window bytewise dependless on the item size. This allows you to change the alignment. Shortcuts **Clrl+LeftArrow** and **Ctrl+RightArrow** scroll windows left and right by columns.

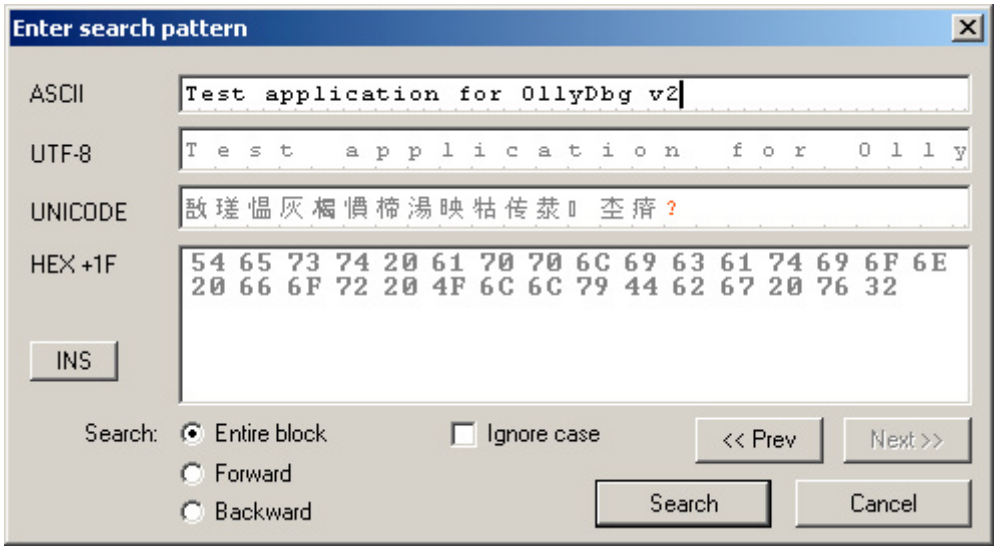
When option **Dump | Underline fixups** is active, OllyDbg underlines fixups in the hexadecimal dumps. Fixup is a location of the adjustable address. If executable file or DLL loads on address that differs from expected, loader will change the value of this address. Be very careful when you place code into the memory occupied by fixup, especially in DLLs! OllyDbg adjusts the contents of fixups back to the default base, but if, for any reasons, module loads at a different base, the code will change (or non-fixuped absolute address will point to the old, now invalid location - I don't know which situation is worser).

Search

Search for binary pattern

OllyDbg gives you many different search possibilities. The simples search is the search for the binary pattern. For example, you want to find the location in *Test.exe* where it creates the main window. This window is titled "**Test application for OllyDbg v2**". As Test.exe is an ASCII application, we expect that this name is ASCII, too. Some compilers place static strings into the code, other may use data section. We don't know this in advance and therefore must check both possibilities, if necessary.

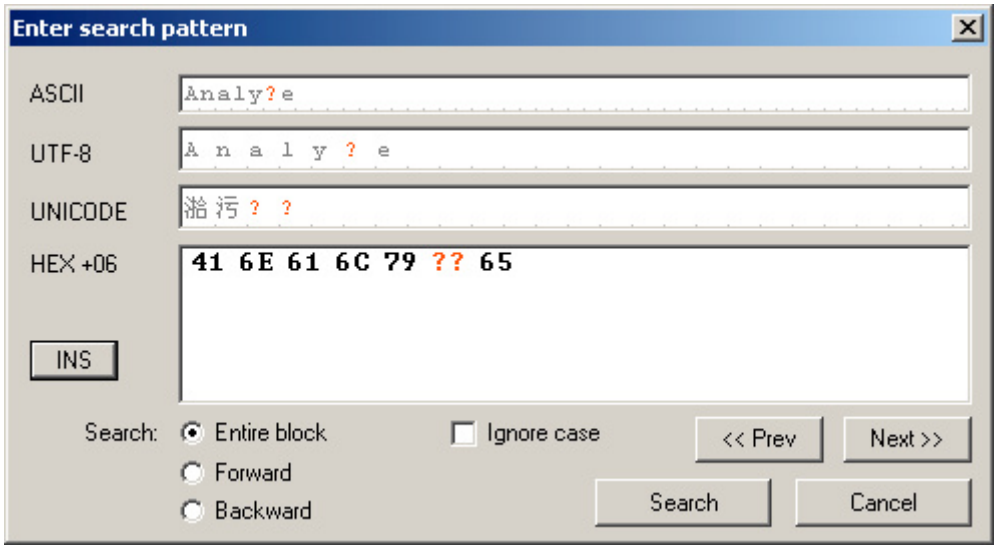
Switch to the CPU Disassembler and press **Ctrl+B** (or choose **Search for | Binary string...** from menu):



The appearing dialog allows you to specify the search string in one of the four formats: as ASCII, multibyte (MBCS) or UNICODE string, or directly as a sequence of the bytes in hexadecimal format. ASCII and MBCS use currently selected codepages. In our example, ASCII control uses default codepage (1252, ANSI - Latin I) and MBCS will encode what you type as UTF-8. Click in the first line (ASCII) and start typing the search pattern. As you write the text, OllyDbg updates all other lines. Where the symbol is invalid, like odd number of bytes in the UNICODE presentation, OllyDbg displays red question mark.

If you select part of the text in one line, OllyDbg also selects all fully or partially selected characters in the remaining controls. For example, if you select the first visible digit 5 in the hex control, OllyDbg will select letter T in ASCII (1 byte) and MBCS (1 to 5 bytes) and first Han letter in the UNICODE line (2 bytes). Note that OllyDbg does not support surrogates (characters encoded by a pair of UNICODE code points): two bytes are always one UNICODE symbol. Words in the range D800..DFFF are interpreted as a separate characters, although they are meaningless in this role.

Hexadecimal line supports masked search. Question mark in hex edit means that the contents of the nibble will be ignored. For example, you want to find the text Analy**s**e - or was it Analy**z**e? Type Analyse in ASCII line and select letter s. Its binary code is 73. Switch to hex editor (mouse or **Ctrl+UpArrow**) and replace 73 by two question marks. This pattern will match both English and American forms of the text:



If you paste to the hex control, it extracts digits and letters from A to F (or a to f) and ignores all other characters. Pasting "W**e** **a**pologi**z**e **f**or th**e** in**c**onv**e**ni**e**n**ce**" will result in EA EF EC EE CE.

After text is entered, select search direction: forward from the beginning of the dump (**Entire block**), forward from the start of the dump selection (**Forward**) or backward from the dump selection (**Backward**). Option "**Ignore case**" is very limited, it works only with standard ASCII characters and ignores localized alphabets. It may lead to errors (false positives) in MBCS and UNICODE.

Now press search. Unfortunately, there is no such string in the code section. Select the very first byte in the CPU Dump and press **Ctrl+L**. This shortcut means "continue the same search in the same direction". (By the way, **Ctrl+Shift+L** means "continue search in the direction opposite to the selected"). This time the text is here. Just to be sure, press **Ctrl+L** again. Status bar blinks with "**Item not found**". The found text is unique.

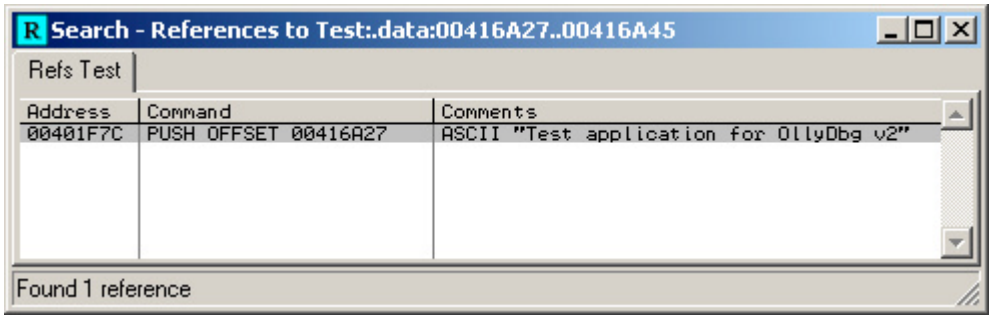
Unfortunately, there is a problem. We have found the text, but we need the command that accesses this text. Don't worry, there is a special option.

Search for references

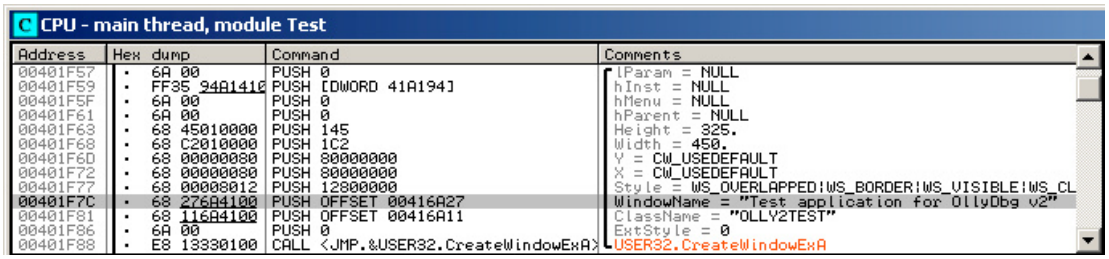
A reference to some constant is a command that includes this constant. For example, all these commands reference address 00402007:

```
MOV EAX,[00402007]
PUSH 00402007
CMP CL,[BYTE 00402007+ESI*4+EDI]
```

We have found the location of the text "**Test application for OllyDbg v2**" in the CPU dump. Select it and from the pop-up menu choose **Find references to | Selected block**. There is exactly one such command:



Doubleclick on this reference. We are inside the call to *CreateWindowEx()*. Perfect, we have found the location where main window is created:



Sometimes search for references may report commands that don't include the constant directly, loke PUSH EDX. This is not an error. Analyser attempts to predict the value of the arguments passed to the functions. The sequence of the commands may be rather complex. For example, search for 417534 will reveal that in the sequence

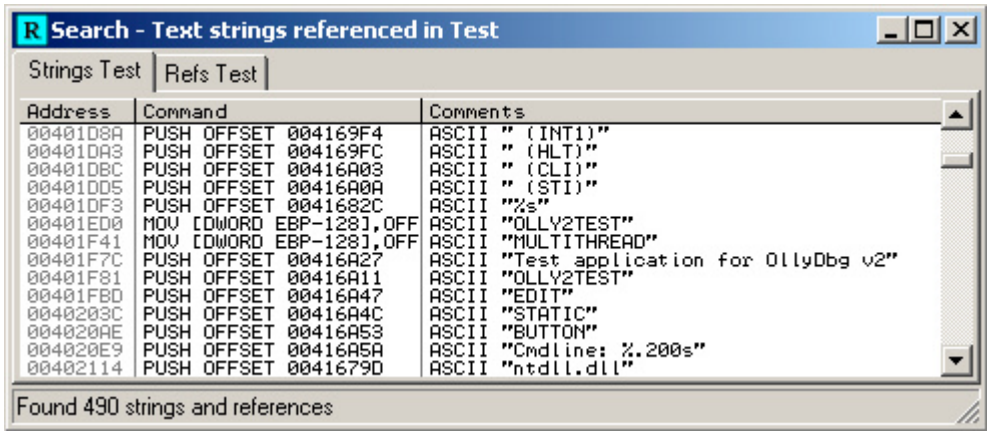
```
MOV EBX, OFFSET 41751C
...         ; Long sequence oif commands that don't change EBX
LEA EDX,[EBX+18]
PUSH EDX
```

register EDX will contain 0041751C+00000018=00417534.

Search for referenced strings

There is a faster way to find the command that references a text, and it works equally good both with ASCII and UNICODE strings.

Switch to the CPU Disassembler and choose **Search for | All referenced strings** from menu:



The list may be pretty long, therefore Search window includes its own search (pop-up menu items **Search for text**, **Search again** and **Search reverse**). After command is found, doubleclick it to follow address in the Disassembler.

Search for strings uses results of the analysis. The recognition of strings is heuristical. By default, ASCII is limited to the Latin subset. To allow international characters, activate option **Strings | Allow diacritical symbols**. Note that this option does not take into account the selected code page.

UNICODE strings are limited to the same character subset as ASCII if option **Strings | Use internal heuristics** is active. Windows API contains its own heuristical analyser *IsTextUnicode()* that should work for many other languages (**Strings | Use IsTextUnicode()**). But care, this function may return false results, or even interprete valid code or ASCII text as UNICODE!

Strings are usually null-terminated. Some languages, like Pascal or Delphi, may use another form, where ASCII text is preceded by the character count. Recognition of Pascal string is controlled by **Strings | Decode pascal-style string constants**.

Note that if you change one of the mentioned options, you must repeat the analysis (**Ctrl+A** in Disassembler).

Search for a constant

Here you can find the commands that include specified constant, both explicitly and implicitly. It is similar to the search for references but is not limited to addresses. For example, search for a constant 1 will find

```
PUSH 1
SHL EAX,1
MOV [BYTE EAX+1],4567
DD 00000001
```

and many other instances where constant 1 is used.

Search for a command or a sequence of commands

You can search for a single CPU command or for a sequence of commands. This is a nontrivial task because 80x86 has many overlapping addressing modes and frequently many encodings for the same command. Let's take, for example, MOV EAX,[EBX]. There are 16 (yes, sixteen!) possible binary encodings:

8B03	- the simplest form
8B43 00	- form without SIB with 1-byte zero displacement
8B83 00000000	- form without SIB with 4-byte displacement
8B0423	- form with SIB byte without scaled index
8B0463	- same
8B04A3	- same
8B04E3	- same
8B4423 00	- SIB byte, 1-byte displacement, no index
8B4463 00	- same
8B44A3 00	- same
8B44E3 00	- same
8B8423 00000000	- SIB byte, 4-byte displacement, no index
8B8463 00000000	- same
8B84A3 00000000	- same
8B84E3 00000000	- same
8B041D 00000000	- SIB byte, 4-byte displacement, scale 1, no base

Each command can be preceded by a prefix DS: (binary code 3E) that doubles the amount of commands.

But this is not a problem for OllyDbg. It will find all listed commands, whether prefixed or not. (If you want to find only commands with segment prefix, you must specify this prefix in the search model).

Commands may include imprecise operands, like "any 32-bit register", "any memory address" or "any operand". For example, MOV EAX,ANY will match MOV EAX,ECX; MOV EAX,12345; MOV EAX,[FS:0] and many other commands.

Imprecise patterns use following pseudooperands:

Keyword	Matches
R8	Any 8-bit register (AL,BL, CL, DL, AH, BH, CH, DH)
R16	Any 16-bit register (AX, BX, CX, DX, SP, BP, SI, DI)
R32	Any 32-bit register (EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI)
SEG	Any segment register (ES, CS, SS, DS, FS, GS)
FPUREG	Any FPU register (ST0..ST7)
MMXREG	Any MMX register (MM0..MM7)
SSEREG	Any SSE register (XMM0..XMM7)
CRREG	Any control register (CR0..CR7)
DRREG	Any debug register (DR0..DR7)
CONST	Any constant
ANY	Any register, constant or memory operand

You can freely combine these keywords in memory addresses, like in these examples:

Memory address	Matches
[CONST]	Any fixed memory location, like [400000]
[R32]	Memory locations with address residing in register, like [ESI]
[R32+1000]	Sum of any 32-bit register and constant 1000, like [EBP+1000]
[R32+CONST]	Sum of any 32-bit register and any offset, like [EAX-4] or [EBP+8]
[ANY]	Any memory operand, like [ESI] or [FS:EAX*8+ESI+1234]

If you are searching for the sequence of commands, it's important to emphasize the interaction of the commands in a sequence. Suppose that you are looking for all comparisons of two memory operands. 80x86 has no such instruction (except CMPS, but it's slow and requires lengthy preparations). Therefore compiler will generate two commands:

```
MOV EAX,[DWORD 408804]
CMP EAX,[DWORD 408808]
```

However, it is possible that compiler will choose ECX instead of EAX, or any other register. To take into account all such cases, OllyDbg has special depending registers:

Register	Meaning
RA, RB	All instances of 32-bit register **RA** in the command or sequence must reference the same register; the same for **RB**; but **RA** and **RB** must be different
R8A, R8B	Same as above, but **R8A** and **R8B** are 8-bit registers
R16A, R16B	Same as above, but **R16A** and **R16B** are 16-bit registers
R32A, R32B	Same as **RA, RB**

Therefore search for XOR RA,RA will find all commands that use XOR to zero 32-bit register, whereas XOR RA,RB will exclude such cases. Correct sequence for the mentioned example is

```
MOV RA,[CONST]
CMP RA,[CONST]
```

Optimizing compiler may insert other commands inbetween:

```
MOV EAX,[DWORD 408804]
MOV ESI,401007
SUB ECX,EAX
CMP EAX,[DWORD 408808]
```

Commands MOV ESI,401007 and SUB ECX,EAX don't change the contents of register EAX. Command SUB ECX,8 modifies EFL, but CMP EAX,[DWORD 408808] doesn't use flags. Therefore two additional commands have no influence on the memory comparison. If option **Allow intermediate commands** in the sequence search dialog is active, they will be ignored. Of course, calls and unconditional jumps are not allowed.

The functioning of the command sequence may be influenced by the jumps from outside. If you want to exclude such sequences, uncheck **Allow jumps from outside**. Other options are the same as in other search dialogs.

There are also several imprecise commands:

Command	Matches
JCC	Any conditional jump (JB, JNE, JAE...)
SETCC	Any conditional set byte command (SETB, SETNE...)
CMOVCC	Any conditional move command (CMOVB, CMOVNE...)
FCMOVCC	Any conditional floating-point move (FCMOVB, FCMOVE...)

Examples:

Pattern	Found commands

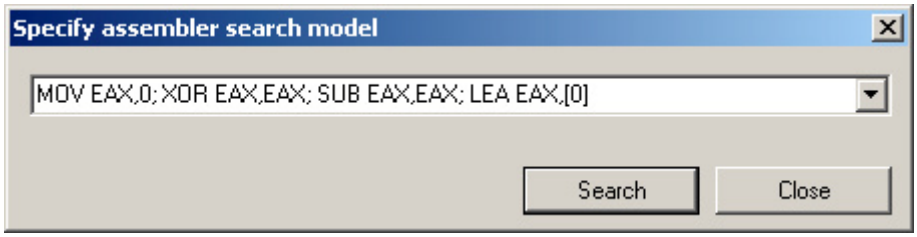
MOV R32,ANY	MOV EBX,EAX
	MOV EAX,ECX
	MOV EAX,[DWORD 4591DB]
	MOV EDX,[DWORD EBP+8]
	MOV EDX,[DWORD EAX*4+EDX]
	MOV EAX,004011BC

ADD R8,CONST	ADD AL,30
	ADD CL,0E0
	ADD DL,7

XOR ANY,ANY	XOR EAX,EAX XOR AX,SI XOR AL,01 XOR ESI,00000088 XOR [DWORD EBX+4],00000002 XOR ECX,[DWORD EBP-12C]
MOV EAX,[ESI+CONST]	MOV EAX,[DWORD ESI+0A0] MOV EAX,[DWORD ESI+18] MOV EAX,[DWORD ESI-30]

Note that in the last line [DWORD ESI-30] is equivalent to [DWORD ESI+0FFFFFFD0].

Each line in the search pattern may include several commands separated by a semicolon. Suppose you want to find all single commands where register EAX is zeroed. Possible (incomplete) solution:

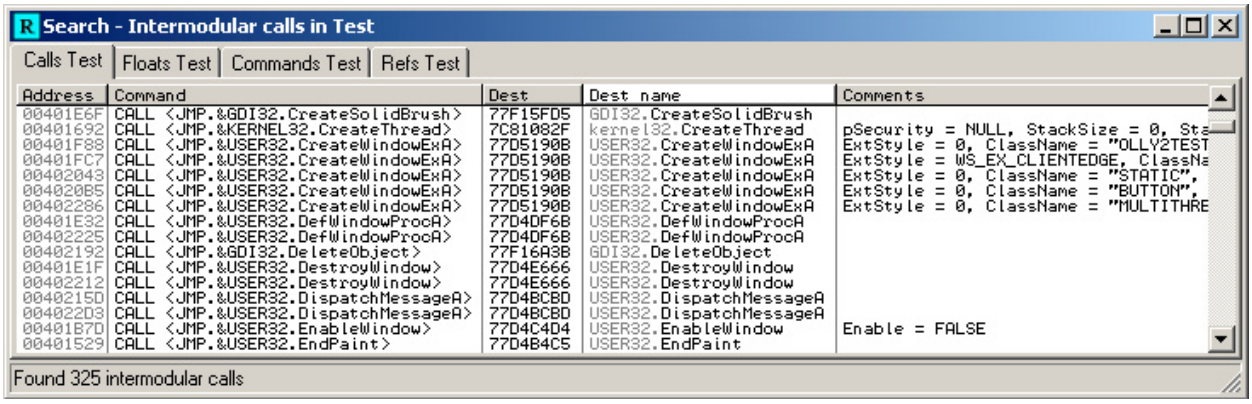


(If you are curious: missing instructions are AND EAX,0, IMUL EAX,0 and MOV EAX,[zeromemory]. SHL EAX,0x20 and friends will not work as expected because shift count in 32-bit mode is restricted to 5 bits).

Search for all items

Many of the search types described above exist in two versions: as a search for a single next item, and as a search for all items.

OllyDbg remembers up to 8 last searches in the tabs of the Search window. These tabs are named by the search function (Calls stays for intermodular calls, Commands for single or multiple commands, Refs for the list of references etc.), followed by the module name:



If there are already 8 tabs in the window, new search will close the rightmost tab. To preserve it, right click on the tab name and choose **Move tab to front**. Middle mouse click on the tab name closes this tab.

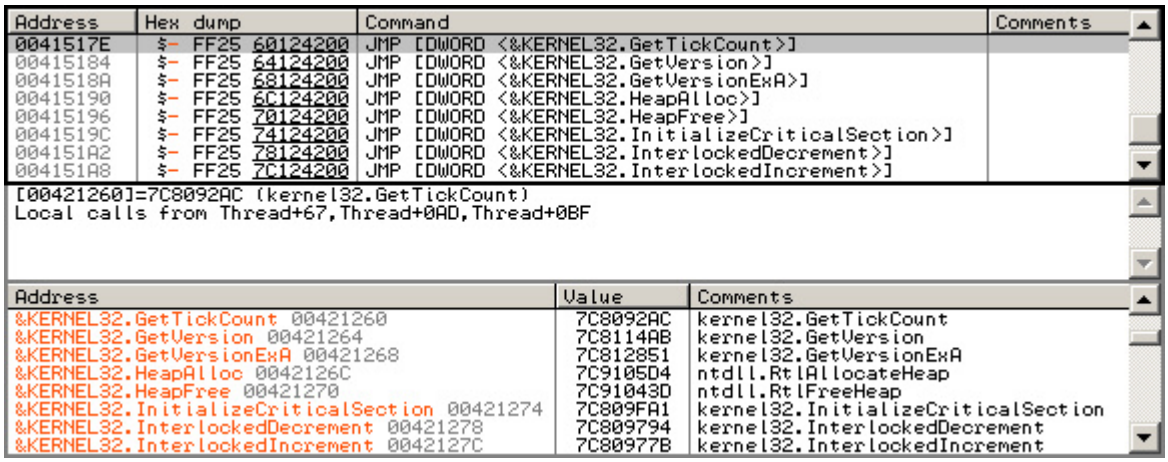
Doubleclick on the line follows command in the Disassembler. Shortcuts **Alt+F7** and **Alt+F8** show all found locations sequentially. By the way, these shortcuts work in the Disassembler, too.

Search for all intermodular calls

This kind of search attempts to find all locations where module calls, directly or indirectly, functions that reside in the different module. An example is shown on the above screenshot. Column with comments lists known parameters passed to the function.

To find function, start typing its name (without module). Search window scrolls to the first function which name begins from the typed characters. Typed names are not case-sensitive.

There are several ways to call imported functions. Borland compilers ask system loader to store real addresses of the imported functions in a table (usually within the section *.idata*, see dump below). At the end of the code section they create a set of indirect jumps, one for each import. Call to import is in fact a call to the indirect jump to this import:



If you want to intercept all static calls from the particular module to the API function, JMP to import is usually the best place.

Note the names that OllyDbg uses to describe these items. Address of the imported function is *module.function*. Memory location in the table of the imports contains reference to the function. To distinguish address of the reference from the address of the function itself, OllyDbg prepends function name with an ampersand: *&module.function*. Call to the function is in fact a call to the jump to this function, but these jumps get no specific labels. To emphasize the fact of the indirect jump and identify the destination, OllyDbg prepends call address with *JMP* and places this construct into the angle brackets: *<JMP.&module.function>*. Finally, disassembled jump will look like

```
CALL <JMP.&KERNEL32.GetModuleHandleA>
```

By the way, have you noticed that address of *HeapAlloc* is labelled as *&KERNEL32.HeapAlloc* but commented as *ntdll.RtlAllocateHeap*? This is an example of export forwarding. For historical reasons, *HeapAlloc()* is declared in kernel32.dll, but its functionality is identical with *RtlAllocateheap()*. It would be possible to write a stub that calls *RtlAllocateHeap()* with the same parameters, or simply jumps to this function. But, to spare linking and execution time, *KERNEL32* tells loader that it must forward this call directly to *ntdll*. For a simple and understandable description of the forwarding I recommend *An In-Depth Look into the Win32 Portable Executable File Format, Part 2* by Matt Pietrek.

Microsoft compilers make intermodular calls a bit differently. The table of the imported destinations is usually placed at the beginning of the code section. Each call to the function is just an indirect call over the import table:

```
CALL [DWORD <&KERNEL32.GetModuleHandleA>]
```

GNU compiler uses calls of both types, even within a single executable file.

Search for calls uses predictions made by the Analyser. If OllyDbg tells you that command CALL EDI is a call to *GetModuleHandleA()*, then there is a MOV EDI,[DWORD <&KERNEL32.GetModuleHandleA>] that precedes this call.

Threads

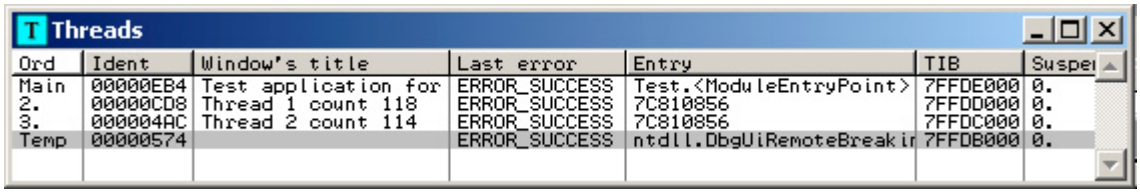
General information

OllyDbg can debug multithreaded applications.

Threads usually have no names. When new thread is created, Windows assign a system-unique identifier to it. Unfortunately, this identifier is different each time you start the application.

To address this problem, OllyDbg numerates all threads as they are created. Main thread of the Debuggee has ordinal number 1. Subsequent thread are numbered 2, 3 and so on. If order in which application creates its threads is fixed, ordinal numbers remain the same in different debugging sessions. Expressions support both identifiers and ordinals.

Sometimes OS creates its own threads, for example, if you pause the application with *DebugBreakProcess()*. OllyDbg labels them as temporary and assigns no (zero) ordinals:



Column '**Window's title**' in the Threads window contains title of the top-level window created by the thread. If thread creates several windows, OllyDbg will randomly select one of them.

Column '**Last error**' shows the most recent error code set by API function. This is the value returned by *GetLastError()*.

You can manually suspend and resume threads. Note that manually suspended threads are not automatically resumed if OllyDbg detaches from the Debuggee.

I have mentioned that threads are usually, but not always, nameless. There is a special service exception 0x406D1388 (MS_VC_EXCEPTION) used by Visual suite and supported by OllyDbg that passes thread names to the debugger. You can also name your threads (pop-up menu of Threads window, **Set symbolic name**). Thread names are not kept between the sessions.

Stepping in multithreaded applications

There is a caveat you must be aware when debugging multithreaded applications. All kinds of stepping, like step over, run trace, hit trace, or execute till return are executed within the current thread. Imagine the following scenario: function xxx posts signal to the different thread and waits for answer. You step over the call to xxx. OllyDbg suspends all threads except for the current, sets temporary breakpoint on the command following call and continues execution. xxx posts signal, waits for answer, and... nothing happens, because thread that processes this signal is paused. If this happens, pause program and run it in the all-threads mode, or manually reswitch to the different thread.

Exception handlers

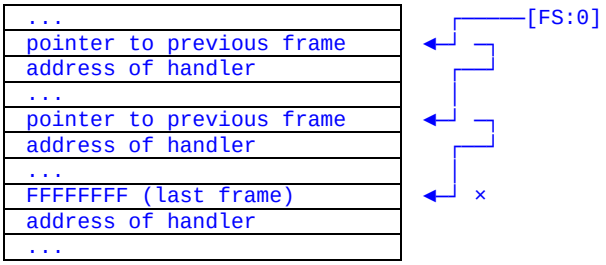
There are two exception handling mechanisms integrated into the Windows: structured exception handling (SEH) and vectored exception handling (VEH). In both cases, exceptions handlers are organized into the linked chains.

SEH handling is thread-dependent. Each thread must install its own handlers separately. Links of the SEH chain are called frames and are kept on the stack. The most recently installed frame is pointed to by the [DWORD FS:0]. (In the flat Win32 memory model selectors CS, DS, ES and DS cover the whole logical memory and have base 0. FS is an exception, its base equals to the address of the thread's data block, also known as a thread's information block, or TIB).

SEH frames roughly correspond to the __*try* ... __*except* blocks in C++. When compiler encounters __*try*, it inserts

```
PUSH <address of handler>  ; Function that processes exception
PUSH [DWORD FS:0]          ; Address of previous SEH frame
MOV [DWORD FS:0],ESP       ; Update SEH frame pointer in TIB
```

or equivalent command sequence. After several handlers are installed, stack has following structure:



To remove last installed handler, thread may execute

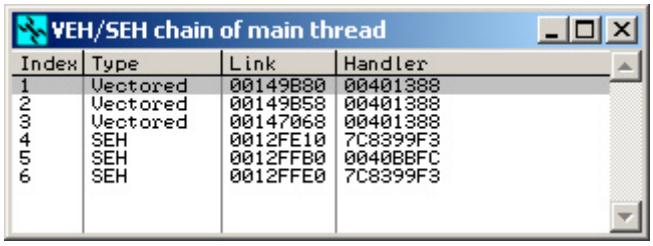
```
POP EDX                    ; Address of previous SEH frame
POP ECX                    ; Address of handler (discarded)
MOV [DWORD FS:0],EDX       ; Update SEH frame pointer in TIB
```

Of course, registers may be different. Instead of POP ECX, compiler may emit ADD ESP,4; instead of [DWORD FS:0] - [DWORD FS:EAX] and precede sequence with XOR EAX,EAX etc.

When exception occurs, Windows walk the chain and asks each handler whether it can handle this event. If handler returns *EXCEPTION_CONTINUE_SEARCH*, Windows try next handler and so on, until handler is found or end of chain is reached, in which case Windows call unhandled exception filter. (Default UEF usually terminates the application).

Vectored exception handling is not available under Windows ME or Windows 2000. Vectored handlers have precedence over SEH and are thread-independent. Process may add handler either to the head or to the tail of the queue. How this queue is managed is not documented. In fact, Windows attempt to hide the addresses of the handlers by scrambling them using process-unique keys and exact location of the queue is not available. Therefore **OllyDbg is able to list VEH handlers only if process was started by the OllyDbg itself**, and only if option **Debugging | Set permanent breakpoint** on system calls was active all the time. Also, it may happen that with the new service pack OllyDbg will lose the ability to trace VEH chain.

Now let's make a quick experiment. Open *Test.exe* in OllyDbg, run it and press button "Set VEH" several times. Pause application and select **View | VEH/SEH chain** from the main menu (or press **Alt+S**):



The handlers are listed in the order in which they will be called when exception occurs (handler with index 1 first).

If you start Test.exe in a standalone mode and attach OllyDbg afterwards, vectored handler addresses will be marked as Invalid.

Expressions and watches

General information

OllyDbg uses expressions in Watches, conditional breakpoints and run trace conditions. Evaluation of expressions is done in two steps. First, OllyDbg compiles expression into the intermediate binary form. On the second step, it calculates its value. The evaluation is usually orders of magnitude faster than compilation, which is vital for breakpoints and trace.

Expressions may be roughly divided in two classes: those used to make a decision (for example, conditional breakpoints) and those used to display data. In the first case you need only one value. In the second case, you may want to see multiple values and expressions at once.

Internally, floating-point numbers have 80-bit precision. All other elements are kept as a 32-bit integer numbers.

Basic elements

Expressions may include:

- Byte registers AL, BL, CL, DL, AH, BH, CH, DH
- Word registers AX, BX, CX, DX, SP, BP, SI, DI
- Doubleword registers EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI
- Segment registers CS, DS, ES, SS, FS, GS
- FPU registers ST0, ST1, ST2, ST3, ST4, ST5, ST6, ST7 or alternative forms ST, ST(0), ST(1), ST(2), ST(3), ST(4), ST(5), ST(6), ST(7)
- Index pointer EIP
- CPU flags EFL or alternative form FLAGS
- FPU registers FST and FCW
- SSE register MXCSR
- Simple labels, like *GetWindowTextA* or *userdefinedlabel*
- Known constants, like *WM_PAINT* or *ERROR_FILE_NOT_FOUND*
- Labels with prepended module name, like user32.GetWindowTextA
- Immediate integer numbers, like ABCDEF01, 123, 0x123 (all hexadecimal) or 123. (decimal)
- Immediate floating point numbers, like 123.456e-33
- String constants, like "String" (used in comparisons)
- Parameters %A and %B
- Thread identifier %THR (or alternative form %THREAD)
- Ordinal number of the current thread %ORD (or alternative form %ORDINAL)
- Contents of memory (requires square brackets [ ], see detailed description below)

Elements are processed in the listed order. Therefore AH is always interpreted as a 8-bit register and not as a hexadecimal constant 0Ah. For the same reason, label EBP can't be used in expressions (but *mymodule.EBP* would be OK). If there is a label named ABCD, ABCD will be interpreted as address of this label and not as the hexadecimal number 0x0000ABCD.

Contents of memory

To access memory, take address into the square brackets and optionally specify type and size of the item. OllyDbg supports following modifiers:

Modifier	How the contents of memory is interpreted
BYTE	Unsigned 8-bit integer
CHAR	Signed 8-bit integer
WORD	Unsigned 16-bit integer
SHORT	Signed 16-bit integer
DWORD	Unsigned 32-bit integer (default)
INT, LONG	Signed 32-bit integer
FLOAT	32-bit floating-point number
DOUBLE	64-bit floating-point number
LONG DOUBLE	80-bit floating-point number
ASCII	Address inside the brackets will be interpreted as a pointer to ASCII string, but numerical value of the expression remains unchanged
UNICODE	Address inside the brackets will be interpreted as a pointer to UNICODE string, but numerical value of the expression remains unchanged

Syntax rules are relatively loose. The following five expressions have the same meaning:

```
[400000+EAX*32]                ; Note that DWORD is assumed by default
DWORD [400000+EAX*32]
[DWORD 32*EAX+400000]
DS:[DWORD 32*EAX+400000]
[DWORD DS:32*EAX+400000]
```

Contents of memory may be used to address another memory. Assume that ppi is declared as *int **ppi* (name is exported as *_ppi*) and we want to monitor the pointed integer number ***ppi*. The correct expression is

```
[INT [[_ppi]]]
```

Note that we need three pairs of brackets. In details:

_ppi	is the address of the variable (*&ppi*)
[_ppi]	is the value of the variable *ppi*, which is the pointer to the pointer to integer
[[_ppi]]	is the value of **ppi*, which is the pointer to integer
[[[_ppi]]]	is the pointed integer number ***ppi*
[INT [[_ppi]]]	asks OllyDbg to interprete ***ppi* as an integer

One more example. We have array *RECT rect[4][4]* and want to inspect *rect[2][3].right*. Structure *RECT* is declared as

```
typedef struct tagRECT {
  LONG    left;
  LONG    top;
  LONG    right;
  LONG    bottom;
} RECT;
```

Let's begin. The size of the structure *RECT* is sizeof(*RECT*) = 4*sizeof(*LONG*) = 16 bytes. *_rect* is the address of the first block of four *RECT* structures in the memory. To get pointer to *rect[2]*, we must

advance this address by 2*4*sizeof(*RECT*) bytes: _rect+2*4*16. To get pointer to *rect[2][3]*, we must move our pointer by 3 strucures: _rect+2*4*16+3*16. Now we need *right*, which is the third doubleword in the structure *RECT*: _rect+2*4*16+3*16+2*4. We have calculated the address of rect[2][3].right. To get the contents of pointed memory, we must take this address into brackets. Final expression will be

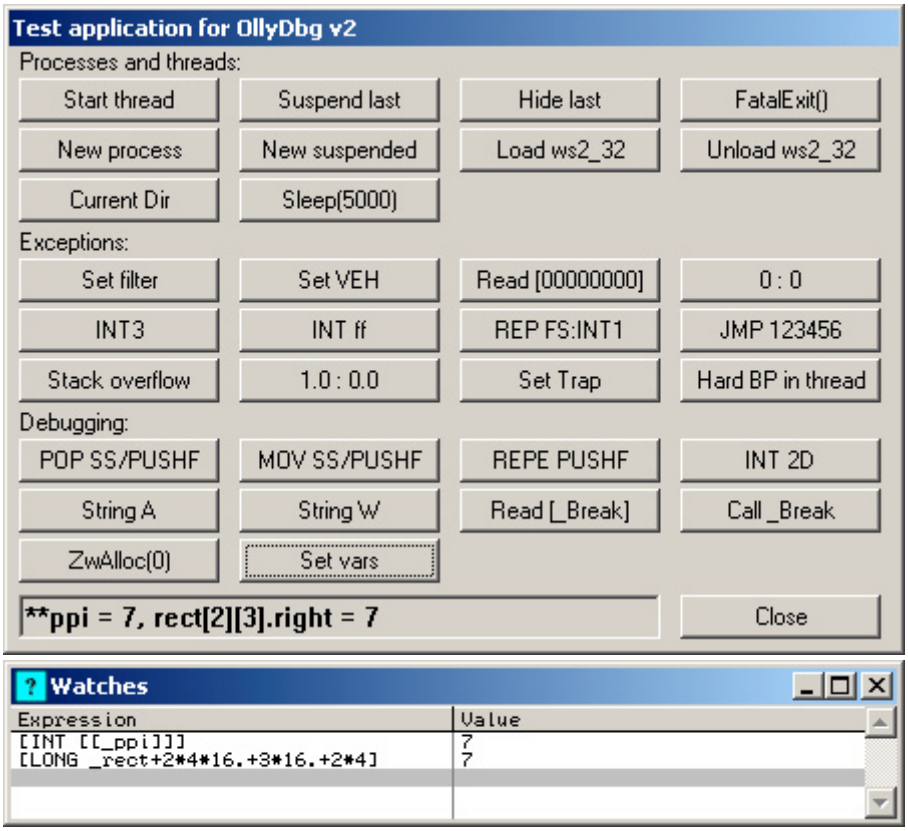
[LONG _rect+2*4*16+3*16+2*4] - *wrong!*

Wait! Do you see the error? By default, OllyDbg interpretes all integers as hexadecimal numbers, therefore the effective size of the structure *RECT* in this expression is $16_{16} = 22_{10}$ bytes, which is not correct. What to do? If number is decimal, append decimal point. The correct expression is

[LONG _rect+2*4*16.+3*16.+2*4]

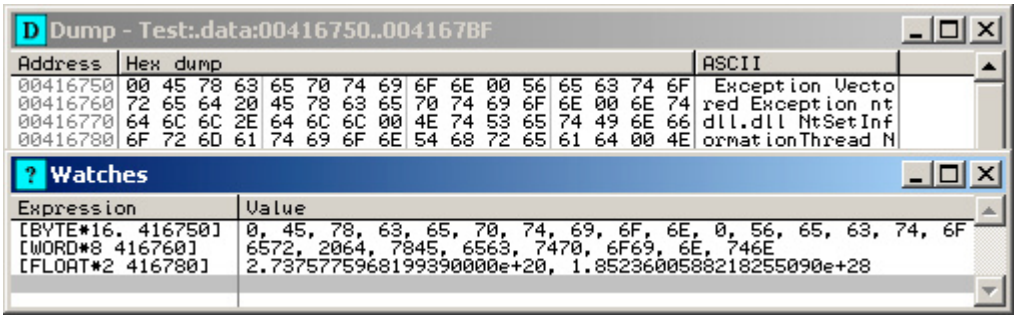
Constants 2, 3 and 4 are the same in both bases, therefore decimal point is not necessary.

Test.exe declares both *_ppi* and *_rect* as described above. To change these variables, press button "**Set vars**":



Note that watches are updated only on some event, for example execution pause or redraw. If you need to constantly monitor the variables, activate **Appearance | Autoupdate** in the pop-up menu. If autoupdate is active, OllyDbg will periodically update the contents of the affected window. Default interval is 1 second, it can be changed in Options (**Appearance | Autoupdate interval**).

You may display up to 16 consecutive memory locations by specifying the repeat count. It is allowed only in the outermost memory bracket and has form [*modifier*repcount address*]. Modifier is obligatory, standalone repeat counts like [*10 ESI] are treated as syntax errors:



Signed and unsigned data

By default, all integer variables and constants are unsigned. Floating-point items are always signed. To interprete item as signed, prepend it with unary sign ('+' or '-') or with modifier SIGNED. To convert signed item to unsigned, use UNSIGNED. Examples:

Unsigned	Signed
0xFFFFFFFF	+0xFFFFFFFF (-1.)
EAX	+EAX
ECX	SIGNED ECX
[DWORD 4F5024]	[SIGNED DWORD 4F5024]

Operations

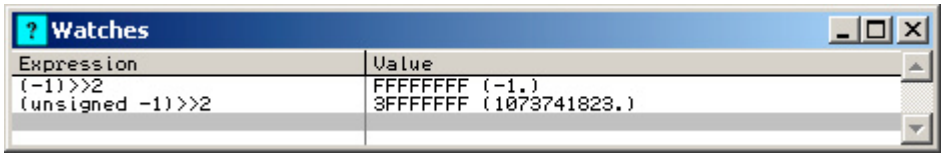
Arithmetic, logical and comparison operations supported by OllyDbg form a subset of the C language operations and have similar precedence:

Operation	Meaning
+ - ! ~	Unary plus, arithmetic negation, logical negation, bitwise negation (highest priority)
*** / %**	Multiplication, division, remainder
+ -	Addition, subtraction
<< >>	Shift left, shift right
< <= > >=	Less than, les than or equal, greater than, greater than or equal
== !=	Equal, not equal
&	Bitwise AND
^	Biwise XOR
|	Bitwise OR
&&	Logical AND
||	Logical OR (lowest priority)

The complexity of expressions and nesting of parentheses and memory brackets are limited only by the maximal allowed length of the expression (255 characters).

If two operands of the binary operation have different types, OllyDbg converts them to the higher type from the following set: (float, unsigned, signed). The only exception are the shifts (<< and >>): if left operand (binary number) is signed and right operand (shift count) is unsigned, OllyDbg makes signed shift

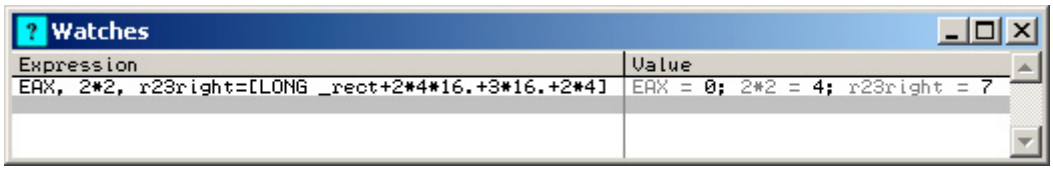
and result remains signed. Have a look:



In the expression (-1)>>2, -1 is a signed number (see rules above) with the binary value 0xFFFFFFFF. Therefore OllyDbg makes signed shifts, and in each shift cycle the most significant bit remains unchanged (in our case 1). In the second expression left operand is unsigned, therefore most significant bit is set to 0.

Multiple expressions

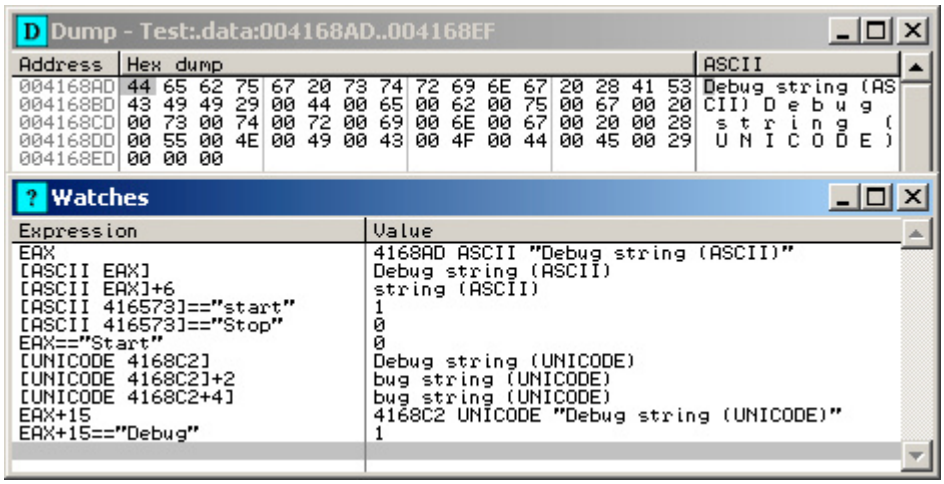
If expression is used only to display data, it may contain several subexpressions in the form text1=expr1,text2=expr2... Number of subexpressions is limited to 16. Explanations (text1, text2) are ignored during the evaluation and just label the results. If explanation is absent, OllyDbg creates one automatically. Example:



String operations

Expressions may contain strings and string constants. Strings have form [*ASCII address*] or [*UNICODE address*], where *address* is an expression that specifies address of the first character.

Operations with strings are very limited. You may add or subtract integer to the string (C rules apply): [UNICODE EAX]+2 is equivalent to [UNICODE EAX+4], because one UNICODE character is 2 bytes long. You may compare string with a string constant, ignoring case: [ASCII ESI]=="ABC". Comparison is limited to the length of the string constant, so this expression is true if ESI points to "ABCDE", "aBc" or "abc---" and false if ESI points to "AB" or "AABC". You may also compare a pointer (doubleword) with the string constant: ESI=="ABC". Such comparison is very similar to the previous, with the difference that you don't need to specify string type - OllyDbg first assumes that ESI is a pointer to ASCII string, and if comparison fails, tries UNICODE. Expression ESI=="ABC" is true if ESI points to ASCII "Abcde" or UNICODE "abc".



Analysis

OllyDbg is an analysing debugger. For each module (executable file or DLL) it attempts to separate code from data, recognize procedures, locate embedded strings and switch tables, determine loops and switches, find function calls and decode their arguments, and even predict value of registers during the execution.

This task is not simple. Some of the existing compilers are highly optimizing and use different tricks to accelerate the code. It's impossible to take them all into account. Therefore the analyser is not limited to some particular compiler and tries to use generic rules that work equally good with any code.

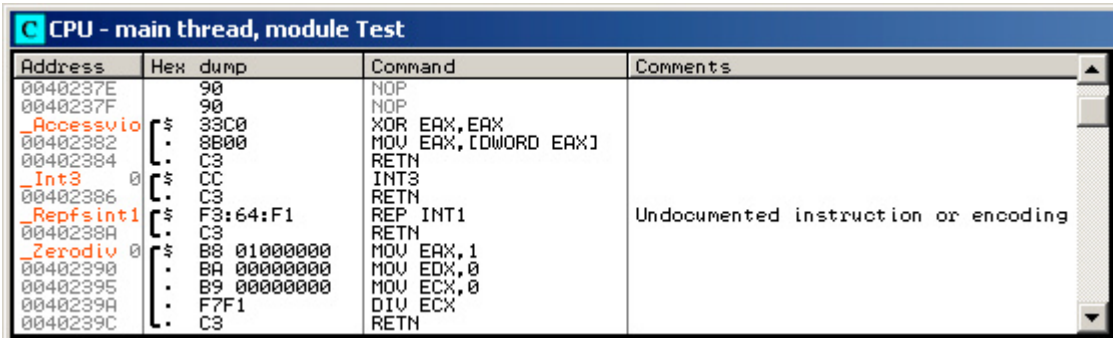
How is it possible at all? OllyDbg makes 12 passes through the program, each time gathering information that will be used on the next steps. For example, on the first pass it disassembles every possible address within the code sections and checks whether this can be the first byte of the command. Valid command can not start on fixup, it can't be jump to the non-existing memory etc. Additionally it counts all found calls to every destination. Of course, some of these calls are artefacts, but it's unlikely that two wrong calls will point to the same command, and almost impossible that there are three of them. So if there are three or more calls to the same address, the analyser is sure that this address is entry point of some frequently used subroutine. On the second pass OllyDbg uses found entries as starting points for the code walk and determines other entries, and so on. In this way I locate commands that are 99.9% sure. Some bytes, however, are not in this chain. I probe them with 20 highly efficient heuristical methods. They are not as reliable, and analyser can make errors, but the number of errors decreases with each release.

Procedures

Procedure is defined as a contiguous piece of code where, starting from the entry point, one can reach, at least theoretically, all other commands (except for NOPs or similar spaceholders that fill alignment gaps). Strict procedure has exactly one entry point and at least one return. If you select fuzzy mode, any more or less consistent piece of code will be considered separate procedure.

Modern compilers perform global code optimizations that may split procedure in several parts. In such case fuzzy mode is especially useful. The probability of misinterpretation, however, is rather high.

Procedures are marked by the long fat bracket in the dump column. Dollar sign (**$)** to the right marks call destinations, sign "greater than" (**>)** - jump destinations, point (•) – other analysed commands. Four procedures on the picture below implement the functionality of the buttons "**Read [000000]**", "**INT3**", "**REP FS:INT1**" and "**0 : 0**"in the test application *Test.exe*:



Stack variables

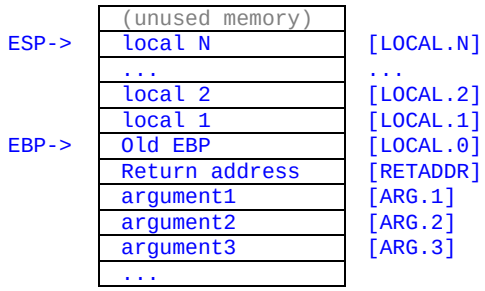
Usually call to the function with several stack arguments looks like this (assuming all doubleword arguments):

```
...
PUSH argument3
PUSH argument2
PUSH argument1
CALL F
...
```

Called function creates new stack frame (not always!) and allocates N doublewords of local memory on the stack:

```
F: PUSH EBP
   MOV EBP,ESP
   SUB ESP,N*4
   ...
```

After these two sequences are executed, stack will have the following layout:



ARG.1 marks the address of the first function argument on the stack, [ARG.1] - its contents, ARG.2 - address of the second argument and so on. LOCAL.0 is the address of the doubleword immediately preceding return address. If called function creates standard stack frame, then [LOCAL.0] contains preserved old ESP value and local data begins with LOCAL.1, otherwise local data begins with LOCAL.0. Note that ARGs and LOCALs have decimal indices - an exception justified by the point in the notation.

Some subroutines tamper with the return address on the stack. If OllyDbg detects access to this location, it marks it as [RETADDR].

When subroutine uses standard stack frame, register EBP serves as a frame pointer. LOCAL.1 is then simply a EBP-4, LOCAL.2 is EBP-8, ARG.1 is EBP+8 etc. Modern optimizing compilers prefer to use EBP as a general-purpose register and address arguments and locals over ESP. Of course, they must keep trace of all ESP modifications. Have a look at the following code:

```
F: MOV EAX,[ESP+8]      ; ESP=RETADDR
   PUSH ESI             ; ESP=RETADDR
   MOV ESI,[ESP+8]      ; ESP=RETADDR-4
```

When procedure is called (address F:), ESP points to the return address on the stack. Two doublewords below is the second argument, ARG.2. Command PUSH decrements ESP by 4. Therefore the last line accesses now ARG.1. The code is equivalent to:

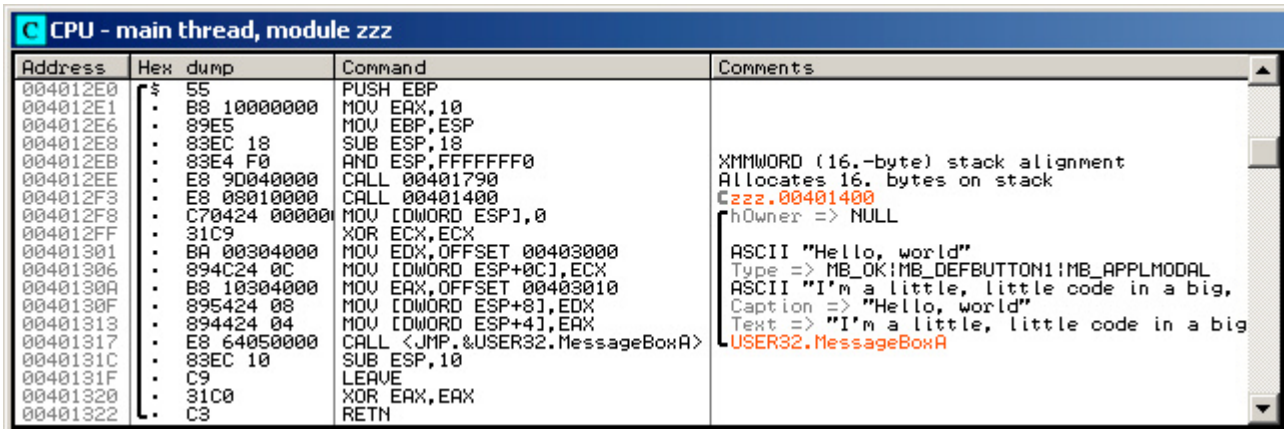
```
F: MOV EAX,[ARG.2]
   PUSH ESI
   MOV ESI,[ARG.1]
```

Of course, analyser makes this for you. Keeping trace of ESP, however, takes plenty of memory. If memory is low, you may turn off the ESP trace for system DLLs. As a negative effect, stack walk will get unreliable.

Some compilers do not push each argument separately. Instead, they allocate memory on the stack (SUB ESP,nnn) and then write arguments using ESP as index. First doubleword argument is [ESP], second - [ESP+4] and so on.

This sample program was translated with MinGW (gcc).

```
int main() {
  MessageBox(NULL,"I'm a little, little code in a big, big world...",
    "Hello, world",MB_OK);
  return 0;
}
```



Note the following: The order of arguments for *MessageBox()* is *(hOwner*, *Text*, *Caption*, *Type)*. MinGW has changed this order. Still, OllyDbg was able to recognize the arguments.

Switches and cascaded IFs

To implement a switch, many compilers load switch variable into some register and then subtract parts of it, like in the following sequence:

```
MOV EDX,<switch variable>
SUB EDX,100
JB DEFAULTCASE
JE CASE100        ; Case 100
DEC EDX
JNE DEFAULTCASE
...               ; Case 101
```

This sequence may also include one- and two-stage switch tables, direct comparisons, optimizations and other stuff. If you are deep enough in the tree of comparisons and jumps, it's very hard to say which case is it. OllyDbg does it for you. It marks all cases, including default, and even attempts to suggest the meaning of cases, like 'A', WM_PAINT or EXCEPTION_ACCESS_VIOLATION. If sequence of commands doesn't modify register (i.e. consists only of comparisons), then this is probably not a switch, but a cascaded if operator:

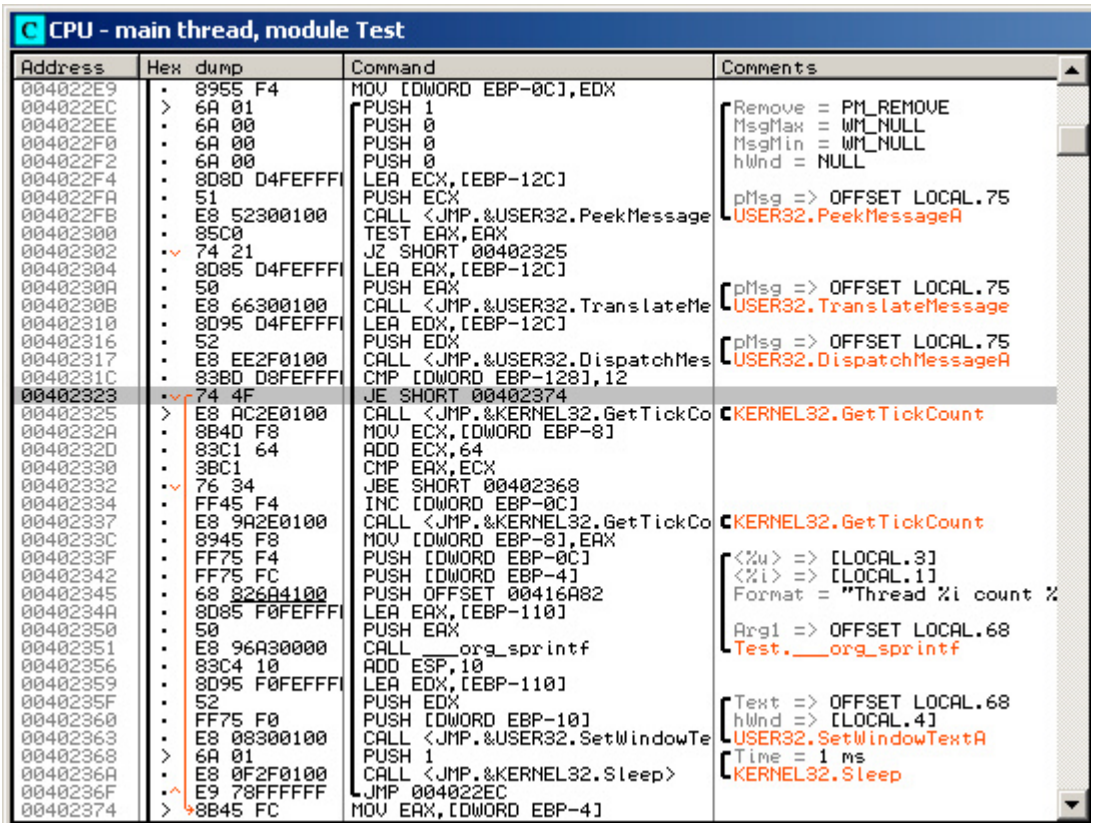
```
if (i==100) {...}
else if (i==101) {...}
else if (i==102) {...}
```

Loops

Loop is a closed continuous sequence of commands where last command is a jump to the first. Loop must have single entry point and unlimited number of exits. Loops correspond to the high-level operators *do*,

while and *for.* OllyDbg recognizes nested loops of any complexity. Loops are marked by parenthesis in the disassembly. If entry is not the first command in the loop, OllyDbg marks it with a small triangle.

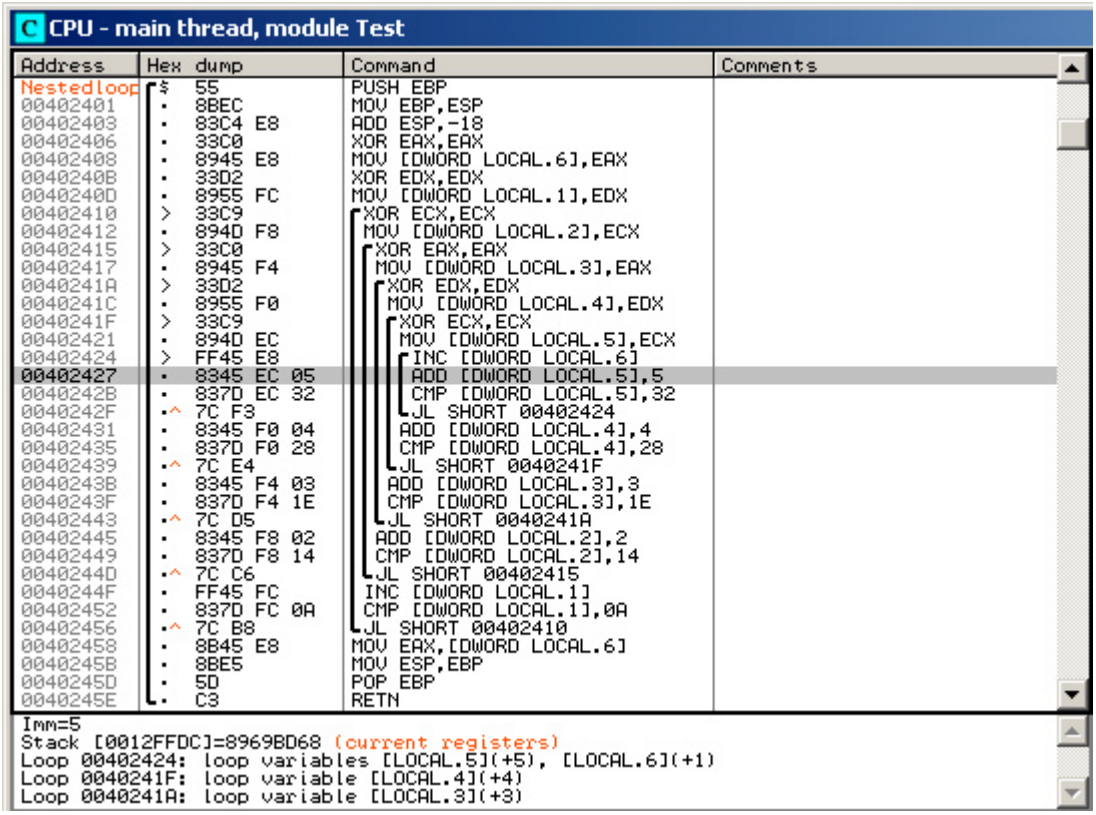
Below is the main loop of the threads created by the test application. Selected command is loop exit. Long red arrow shows its destination:



Note that sprintf() called at address 00401AC1 is a library function used by compiler and is not known as such. But there are other calls somewhere in the code, so analyser was able to decide that its second parameter is a format string, and has decoded the remaining parameters correspondingly. Here is the C code of the loop

```
...
while (1) {
  if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
    if (msg.message==WM_QUIT) break;
  };
  if (GetTickCount()>t+100) {
    u++; t=GetTickCount();
    sprintf(s,"Thread %i count %u",threadindex,u);
    SetWindowText(hw,s);
  };
  Sleep(1);
};
return threadindex;
```

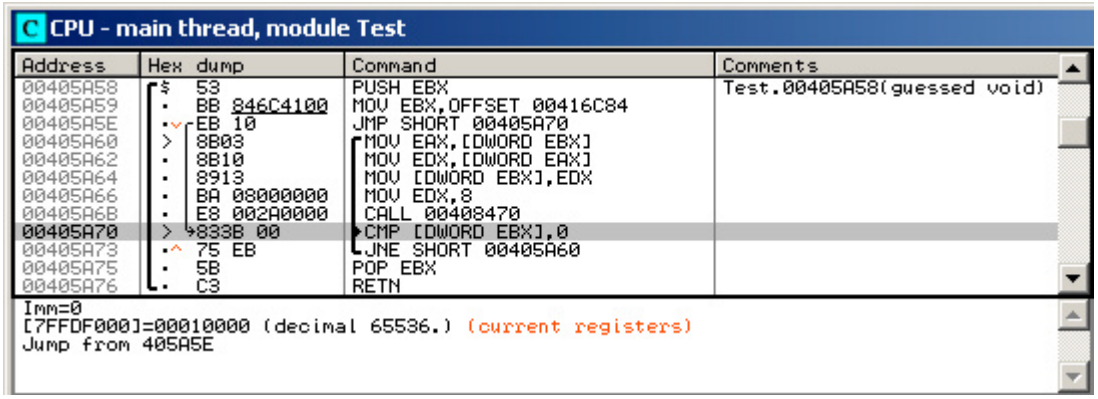
Loops can be nested. Procedure *Nestedloops* in *Test.exe* is an artificial example with 5 nested loops.




```
int _export Nestedloops(void) {
  int i,j,k,l,m,n;
  n=0;
  for (i=0; i<10; i++) {
    for (j=0; j<20; j+=2) {
      for (k=0; k<30; k+=3) {
        for (l=0; l<40; l+=4) {
          for (m=0; m<50; m+=5) {
            n++;
          };
        };
      };
    };
  };
  return n;
};
```

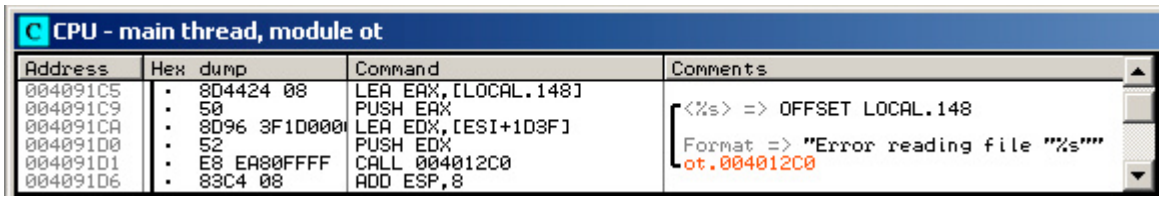
Loop-related comments in the information pane can be interpreted as following: loop that starts at address 00402424 has two variables that increment by a constant on each pass: variable [LOCAL.5] (fifths doubleword of locally allocated data on the stack) increments by 5, variable [LOCAL.6] - by 1. They correspond to variables *m* and *n* in the source code. Variable [LOCAL.4] (*l*) in the loop 0040241F increments by +4, and so on.

If loop entry point is not the first command in the bracket, it is marked with the small triangle:



Prediction of registers

Look at this code snippet:



Analyser has recognized that procedure at address 004012C0 is similar to *printf*: its first argument is a format string. (In fact, this function displays error messages). Such procedures may have variable number of arguments and use C-style conventions for the parameters, namely that calling program must remove arguments from the stack when call returns. Command ADD ESP,8 after the call does exactly this: it pops 8 bytes of data, or two doublewords. The first doubleword is the pointer to format string. Therefore this call must have one additional parameter. Format string indicates the same: it expects one pointer to the string.

String address is the second argument in the call. By stating <%s> => OFFSET LOCAL.148 Ollydbg tells you the following: register EDX at the moment of PUSH EDX will contain address of the local variable 148 (148.-th doubleword preceding return address). Indeed, previous command (LEA EAX,[LOCAL.148]) loads this address to EDX.

Symbol **=>** in comments means "predicted to be equal to". For example, PUSH ECX will push address of the displayed format string... Wait! Operator at 004091CA just adds 0x1D37 to ESI. But where ESI is defined? Somewhere in the code that precedes this call; in fact, this location is more than 300 bytes back. The complete procedure has the following structure:

```
...
MOV ESI,OFFSET Ot.0045E7EC ; Here ESI is defined
...
PUSH ESI
...                        ; Code that uses ESI
POP ESI
...
CALL Ot.xxx
...
CALL OT.yyy
...
LEA EDX,[LOCAL.148]        ; Our code
PUSH EDX
LEA ECX,[ESI+1D37]
PUSH ECX
CALL Ot.004012C0
ADD ESP,8
...
```

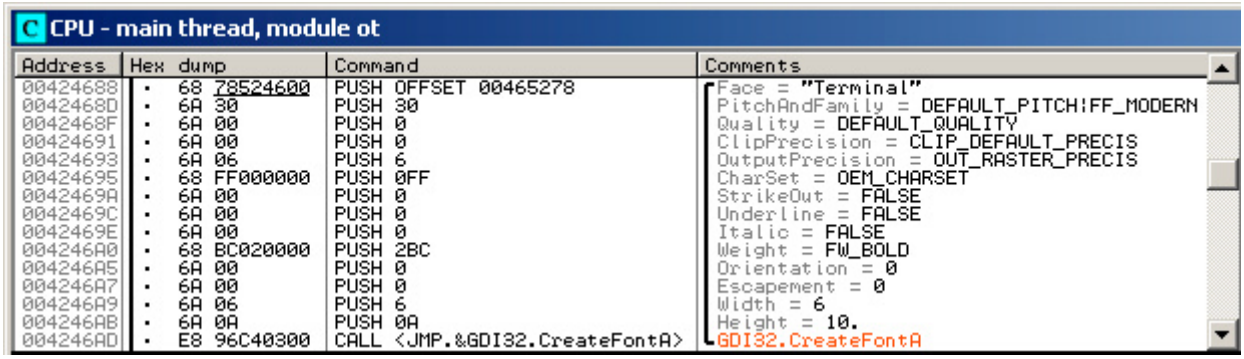
At the beginning, program loads ESI with the address of static data block. This operation is not very meaningful here but would make more sense in the multithreaded applications that use thread local storage. Pair PUSH ESI/POP ESI preserves value of ESI from the modification. Then you see two calls to unknown functions *xxx* and *yyy*. Why analyser is sure that they leave ESI unchanged? Well, either it was able to determine this directly from the code of *xxx* and *yyy*, or you told it by the corresponding analysis option (**Advanced analysis | Unknown functions preserve registers EBX, EBP, ESI and EDI**). All Windows API function and many separately compiled procedures use *stdcall* convention, where functions must preserve registers EBX, EBP, ESI and EDI.

Known API functions

OllyDbg contains descriptions of more than 2300 standard API functions from the following libraries:

- KERNEL32.DLL
- GDI32.DLL
- USER32.DLL
- NTDLL.DLL
- VERSION.DLL
- ADVAPI32.DLL
- SHLWAPI.DLL
- COMDLG32.DLL
- MSVCRT.DLL

It also knows more than 10000 symbolic constants, grouped into 540 types, and can decode them in the code:



You can use known constants in assembler commands and arithmetic expressions. For example, CMP EAX,WM_PAINT is a valid command for the Assembler.

Standard library functions

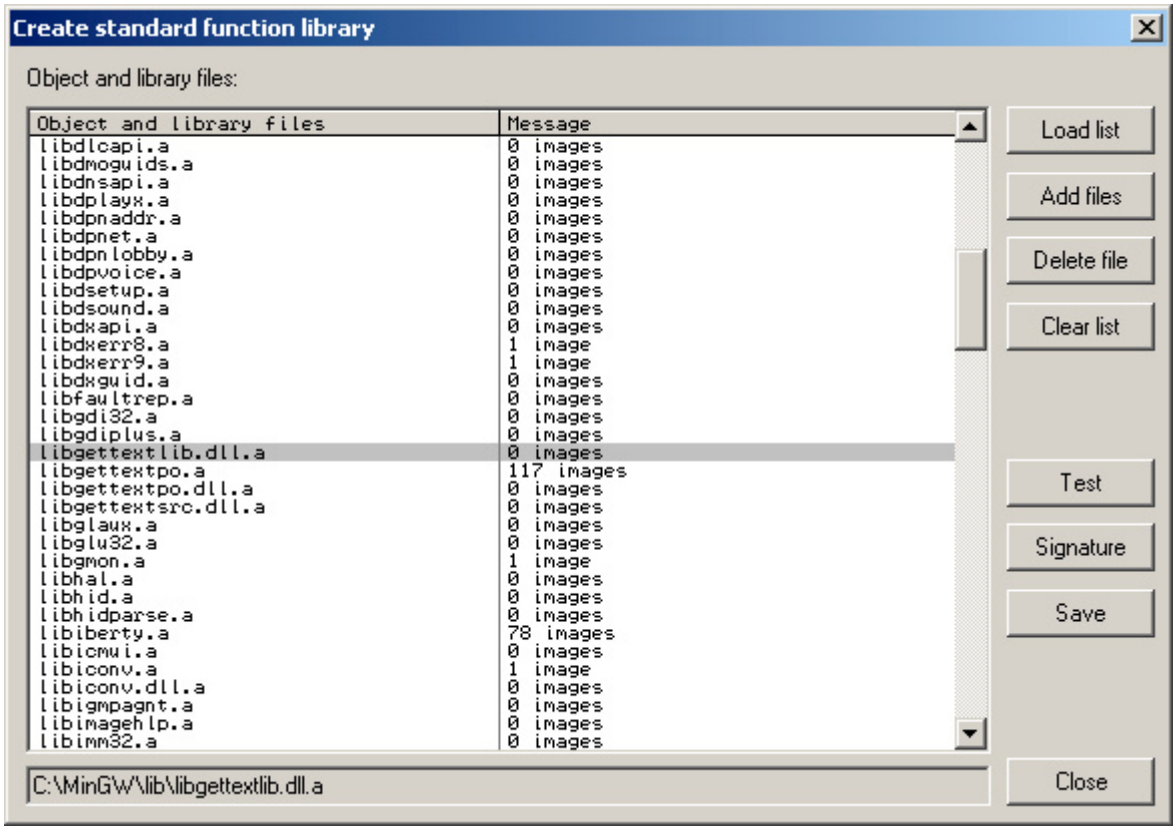
OllyDbg is able to locate standard library functions, like *fopen*, *malloc* or *sprintf*, by scanning the libraries and comparing library code with the code of the debugged program. This is accomplished in two steps.

License agreements wit the IP owners may prevent you from any manipulations with the standard libraries except linking with the code produced by the given compiler. Therefore our example is based on the MinGW compiler, which is distributed under GPL.

On the first step you must select standard libraries supplied with the compiler and create the *.udl* file. OllyDbg supports OMF (Borland, optionally Watcom) and COFF (Microsoft, GNU etc) formats, both as libraries and a single object files. Usually they have extensions *.lib* and *.obj*. GNU suite uses *.o* for object files and *.a* for object libraries. If MinGW was installed with the default parameters, then the main libraries are located under *c:\MinGW\lib*.

Choose **Debug | Create function library** from the main OllyDbg menu. Press **Add files**, navigate to the directory *c:\MinGW\lib* and select all listed files and libraries. Press **Open**. They will appear red in the dialog, emphasizing the fact that they are not yet processed.

Now press **Test**. OllyDbg reads files and checks their contents. This operation is quick, and in the second column you will see the messages like "48 images". This is the number of object files within the library that contain binary code:



How is it possible that file *libgettextlib.dll.a* is more that 1 MB large but contains no images? This file is an import library. When selected, it reports to linker that, say, *_xmlTextWriterFlush* and many other functions are located in the dynamic link library *libgettextlib.dll*. This information is of no use to the OllyDbg.

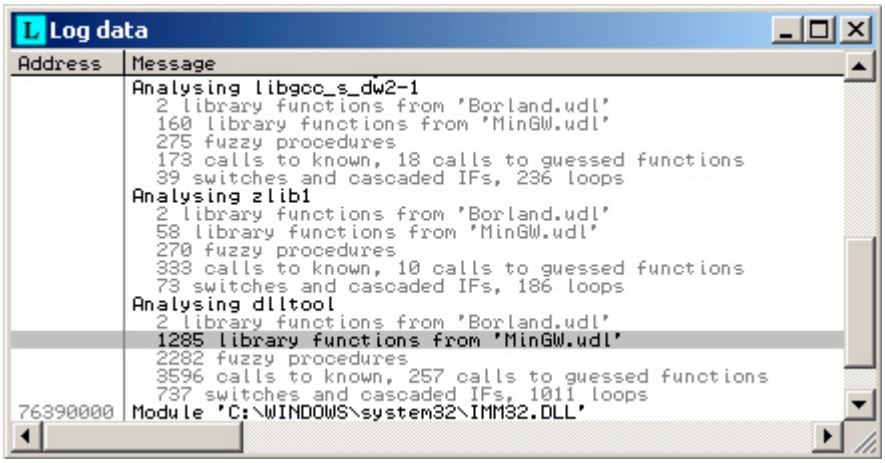
There are also object files that implement low-level functions, like shifts or long divisions. They are located under *c:\MinGW\lib\gcc\mingw32\4.8.1* (here 4.8.1 is the MinGW version). Press **Add files** again and select them, too.

Now you can remove all files with no useful images. But this will not influence analysis time. Therefore: press **Save** and choose subdirectory *udl* (this is where OllyDbg looks for udl libraries by default, can be changed in **Options | Directories**).

Search for standard libraries is relatively fast, but depends on the number of images in the library. It may be wise to not include rarely used libraries into the udl file.

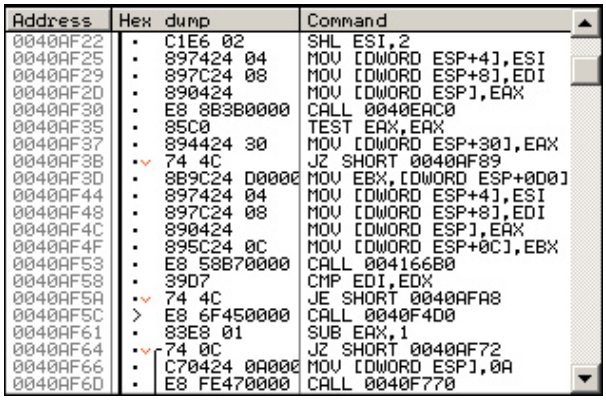
Now the library is prepared. Whether analyser will search for standard functions is controlled by the option **Advanced analysis | Detect standard library functions (*.udl)**.

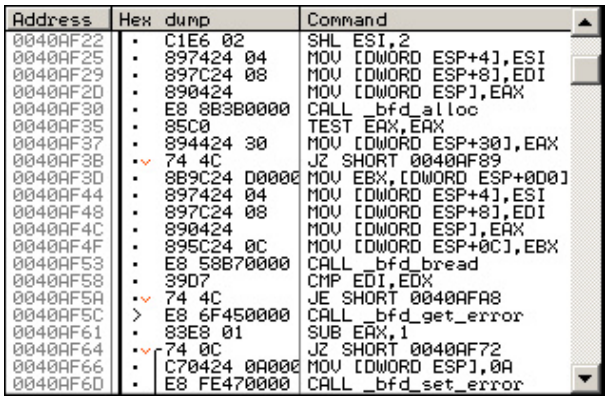
Let's load some file that uses MinGW library, for example MinGW utility *dlltool.exe*. Log window informs us that Analyser has found many standard functions. In the list of names their type is marked as "Analyser". But log also reports that there are two functions from the Borland's library. We are sure that MinGW doesn't use Borland's code, aren't we:



There are really several very simple functions in Borland and MinGW libraries that have the same binary code. For example, they may call standard function from the system DLL. Usually they have identical functionality. On the other hand, standard library may include many identical functions that have (in some sense) different meaning, like standard destructors. To play it safe, OllyDbg does not include ambiguous functions into the udl library.

Search for standard functions on the left screenshot was turned off, on the right - on:



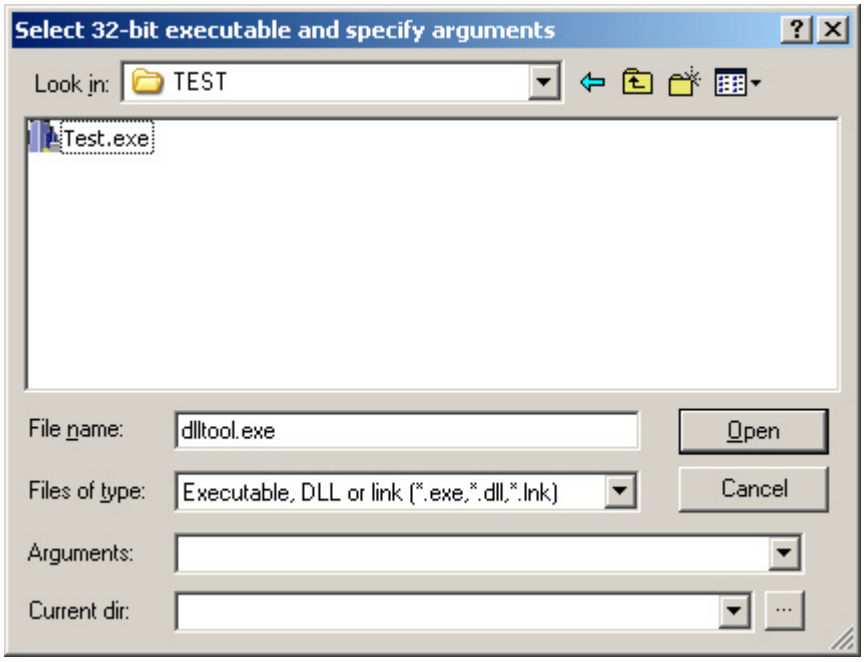


Debugging

Opening the program

The simplest way to open the executable file in OllyDbg is to drag-and-drop it into the main OllyDbg window. OllyDbg supports executables (*.exe*) and link files (.lnk). Debugging of standalone DLLs is also possible, see below.

If you need to specify the command line arguments, select **File | Open...** from the main menu or press **F3**, then type your arguments into the corresponding line. Length of arguments is limited to 1023 characters:



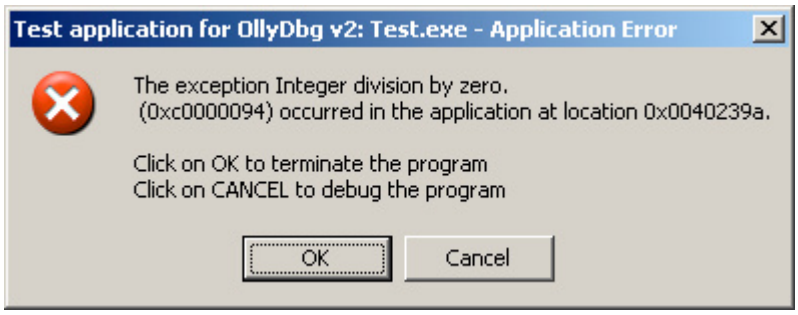
You can instruct OllyDbg where the to pause the newly started application for the first time by setting one of the **Start | When starting application, make first pause at** options:

- **System breakpoint** - application will pause at *ntdll.DbgBreakPoint*;
- **TLS callback** - at the entry point of the first TLS callback defined in the executable file. If there are several callbacks and you want to check them all, set breakpoints manually. Callbacks are named as *<TLS_Callback_1>*, *<TLS_Callback_2>* etc. If there are no TLS callback functions, first pause will be executed at the entry point of the main module.
- **Entry point of main module** - at the module entry point, as defined in the PE header;
- **WinMain** - at the entry point of the function *WinMain()*, if this entry is known to the OllyDbg. If address of *WinMain()* is unknown, first pause will be executed at the entry point of the main module;
- **No pause** - no pause at all, application runs immediately.

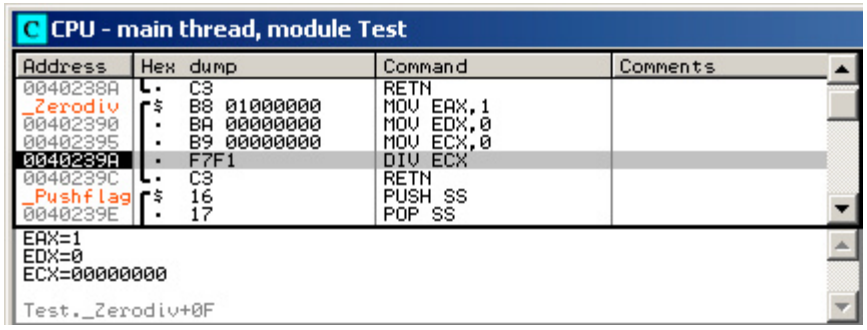
OllyDbg as a just-in-time debugger

You can register OllyDbg as a just-in-time debugger. In the Options dialog open panel **Just-in-time** and press **Set OllyDbg**. Now, if some application crashes, you will be given an option to debug it. (If you uncheck **Confirm before attaching**, all buggy programs will be automatically redirected to OllyDbg - this is probably not what you want on everyday basis).

Let's investigate this possibility. Register OllyDbg, open *Test.exe* in a stand-alone mode and press button "**0 : 0**":



Answer with **Cancel**, and OllyDbg will automagically start with CPU window positioned on the command that caused the exception:



Now you can do anything, as usually. For example, you may point EIP to the next command. Select RETN at 0040239A, choose **New origin here** from the pop-up menu and execute **File | Detach** from the main menu (not available under Windows NT/2000). *Test.exe* continues execution as if nothing happened.

On detach OllyDbg removes all breakpoints. It doesn't resume manually suspended threads.

Attaching to the running processes

You can attach OllyDbg to the running process, provided that you have sufficient privileges. From the main menu, select **File | Attach...** and choose process from the list of running processes.

If main thread is suspended (for example, application was created as *CREATE_SUSPENDED*), OllyDbg will automatically resume it. For different reasons, this is not possible under Windows 2000. Attempt to attach to the suspended application will result in crash.

Attaching to running process is controlled by the options in **Start | When attaching to application, make first pause at**:

- **System breakpoint** - application will pause on the system breakpoint in the temporary thread. While attaching process to debugger, Windows create new thread in the contents of the Debuggee. This thread executes *DbgBreakPoint()*, giving OllyDbg the chance to make all necessary preparations. The thread is marked as temporary by the OllyDbg. Note that under Windows 2000, OllyDbg is unable to recognize this thread as temporary and reports it as an ordinary thread;
- **Application code** - asks to pause application in the main thread at the location that was executing at the moment of attaching. Usually this is *ntdll.dll*;
- **No pause** - application should continue execution as soon as possible.

Debugging of child processes

OllyDbg is a single-process debugger. To debug child processes, it launches new instances of itself, so that each child gets its own copy of the OllyDbg. This is possible only under Windows XP or higher Windows versions, and only if parent process was started by OllyDbg.

Due to the limitations of the Windows, debugging of grandchildren is not supported. That is, if you debug process A and it spawns process B, B will be passed to the OllyDbg. If now B spawns process C, debugger will get no notification and C will run free. Of course, you can attach to the C later.

Debugging of child processes is controlled by the option **Debugging events | Debug child processes**.

Breakpoints

OllyDbg supports three types of breakpoints:

- **Software breakpoints** on code execution;
- **Memory breakpoints** on memory access, writing to memory and/or code execution;
- **Hardware breakpoints** on memory access, writing to memory *or* code execution.

Additionally, you can set breakpoint on access to memory block (not available for DOS-based Windows versions). They were discussed previously.

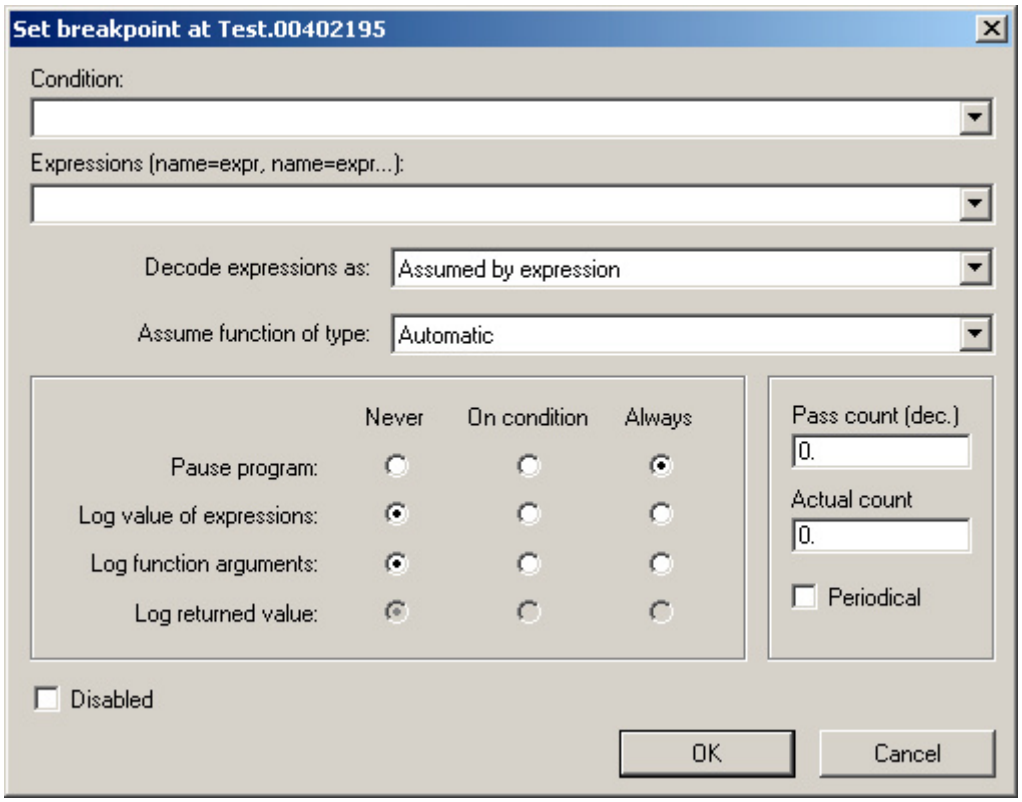
Each type has its own advantages and drawbacks. They are listed in the table below:

Type	**Software**	**Memory**	**Hardware**
Principle of operation	First command byte is replaced by INT3 or 1-byte privileged command (HLT, CLI, INSB etc.)	Change of access protection using *VirtualProtectEx()*	Reprogramming of debug registers using *SetThreadContext()*
Number	Unlimited	Unlimited	4
Conditional / logging	Yes	Yes	Yes
Execution speed (if not hit)	Not influenced	May be very low	Not influenced
If set on the wrong location	May crash anywhere	May crash if pointer to protected memory is passed to kernel	Usually OK
Detection by the application	Very easy (read memory)	Very easy (*VirtualQuery()*)	Easy (*GetThreadContext()*)

Note that Windows protects memory only in chunks of 4096 bytes. Therefore, if memory breakpoint is set within the memory block with frequently accessed variables, OllyDbg will get many false breakpoints, which may significantly slow down the speed of execution. In some cases, run trace with allowed emulation of the commands may be faster! For details, see "Run trace and profiling". Caveat**: avoid setting memory breakpoints on the stack**. If protected address is passed to the system, system call may fail, terminating the application.

Simple breakpoint pauses execution each time the command is executed or memory is accessed. You may also set conditional breakpoints that pause execution only if some condition is met, and conditional logging breakpoints with many additional options. For example, they may protocol some parameters if condition is true without pausing program.

All dialog windows that set breakpoint options have similar structure. As an example, here is the dialog window of the software conditional logging breakpoint:



Logging breakpoints support four actions:

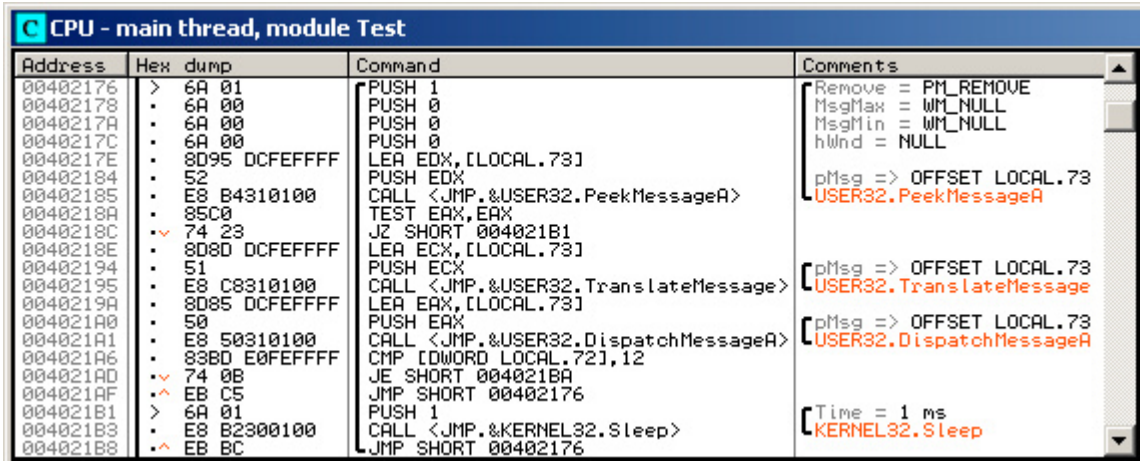
- **Pause program** - application will be paused;
- **Log value of expressions** - expressions specified in the second line will be evaluated and their values protocolled to the Log window. You may use multiple expressions separated by comma and use repeat counts to display consecutive memory locations, see chapter "Expressions" above for details. You may influence the decoding by specifying expression type in the **Decode expression as** control;
- **Log function arguments** - if breakpoint is set on the call to the known or suggested function, or if you specify type of the function explicitly (**Assume function of type**), stack arguments will be decoded and protocolled to the Log window;
- **Log returned value** - if breakpoint is set on the command that immediately follows CALL of known function and this function returns a value, the contents of register EAX will be decoded and protocolled to the Log window.

For each action you have three options:

- **Never** - action is not executed;
- **On condition** - action will be executed only if condition in the first line evaluates to TRUE (non-zero)
- **Always** - action always takes place.

Additionally, you may skip several first breakpoints by specifying the pass count. If **Periodical** is checked, only each **Pass count**-th breakpoint will be processed.

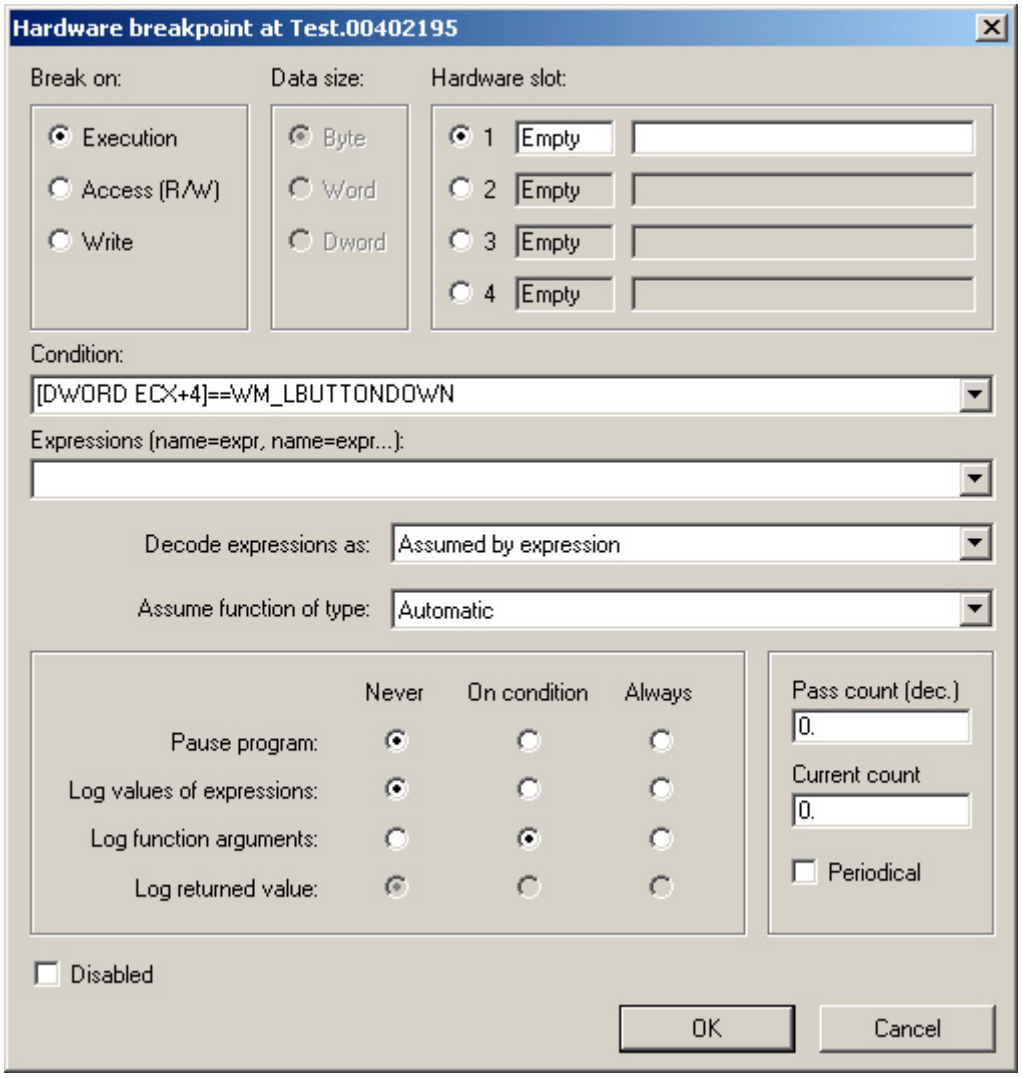
As an example, assume that we want to protocol all messages of type *WM_LBUTTONDOWN* passed to the main thread of the *Test.exe*. Here is the main Windows loop of this application:



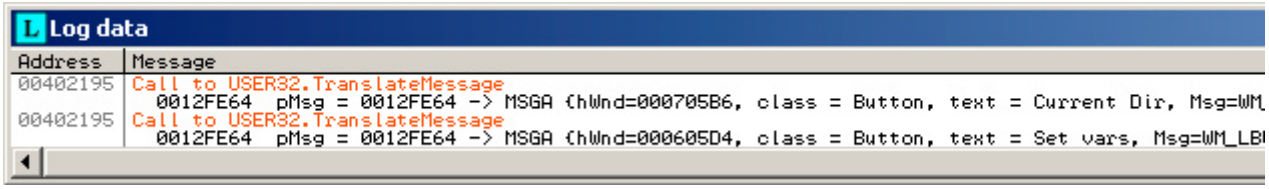
Messages are fetched by the call to *PeekMessageA()*. This API returns 0 if there are no messages. Sequence TEST EAX,EAX; JZ 004021B1 checks this condition and skips if message is missing. Next API, *TranslateMessage()*, takes pointer to the message as its only parameter. Perfect, let's set breakpoint on this call and log function arguments. However, if you try this, OllyDbg will protocol all messages, not only *WM_LBUTTONDOWN*.

Structure *MSGA* passed to *TranslateMessage()* has message code as its second doubleword parameter. Note that *Test.exe* is an ASCII application and therefore API structure *MSG* gets suffix *A* that distinguishes it from the UNICODE counterpart, *MSGW*. Have a look at the disassembly: at the moment of call, register ECX still contains pointer to this structure. A pointer to the structure is the address of its first item. If first item is a doubleword (4 bytes), address of the second item is ECX+4. This item is in the memory. To get its contents, we must take memory address in the square brackets and specify memory size. (As DWORD is a default, memory size is optional). Therefore the correct expression is [DWORD ECX+4]==WM_LBUTTONDOWN. The constant *WM_LBUTTONDOWN* is known, we don't need to search header files to get its value.

Let's use hardware breakpoint. Select call command, right click and choose **Breakpoint | Hardware log...** Hardware breakpoints may trigger on the command execution, on memory access which is not the command execution, or on writing to the memory (but not on their combination!) Choose **Execution**. In this case data size is always 1 byte and second panel is grayed. Now select one of the 4 available hardware breakpoints. In our case all breakpoints are free and it doesn't matter which one will be used:



Confirm your choice, run application and press buttons "**Current dir**" and "**Set vars**". Now have a look into the Log window:

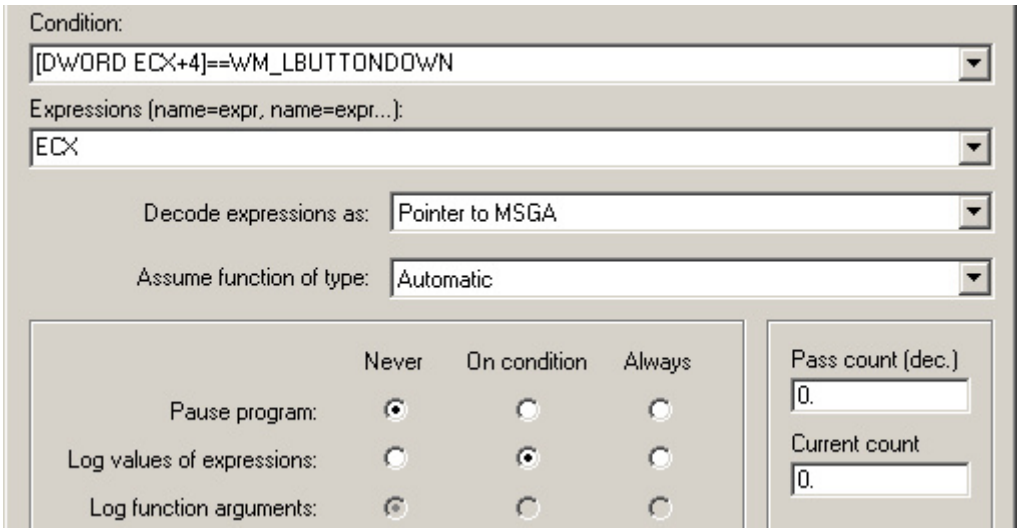


Look at the first entry. Parameter of *TranslateMessage()* is a doubleword containing 0012FE64. This parameter is named *pMsg* and is a pointer to the structure *MSGA*. First element of the structure *MSGA* is *hWnd* - handle of the window of class "*Button*" containing text "*Current dir*". Now comes the second *MSGA* member, message identifier *Msg* which is equal to WM_LBUTTONDOWN, etc., etc.

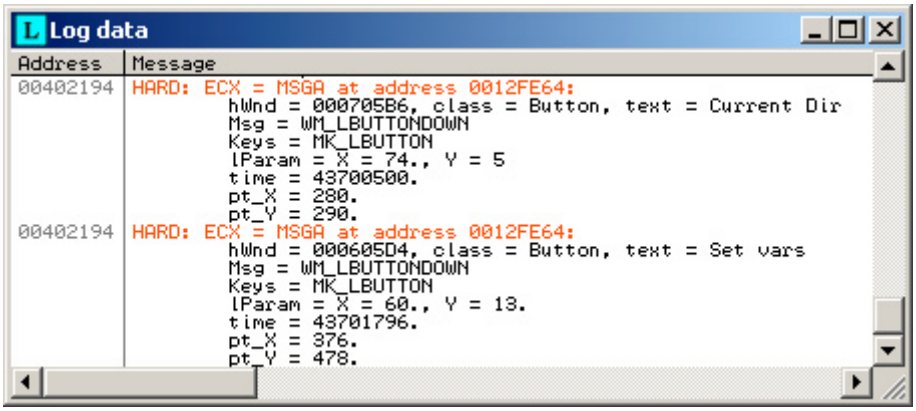
In this example we were lucky because *TranslateMessage()* is a known function (present in the API database). What can one do if this is not the case? Let's try slightly different approach. Have a look at the command sequence:

```
0040218E   LEA ECX,[LOCAL.73]
00402194   PUSH ECX
```

First command loads ECX with the pointer to the structure of type *MSGA*. Remove previous breakpoint and set new at PUSH ECX. Now we must specify the expression to protocol: ECX. This expression must be interpreted as a pointer to *MSGA*. Finally, the value of expression must be logged only on condition which remains the same as in the previous case:



Press the same buttons again and inspect the log. Now it's more readable:



Run trace and profiling

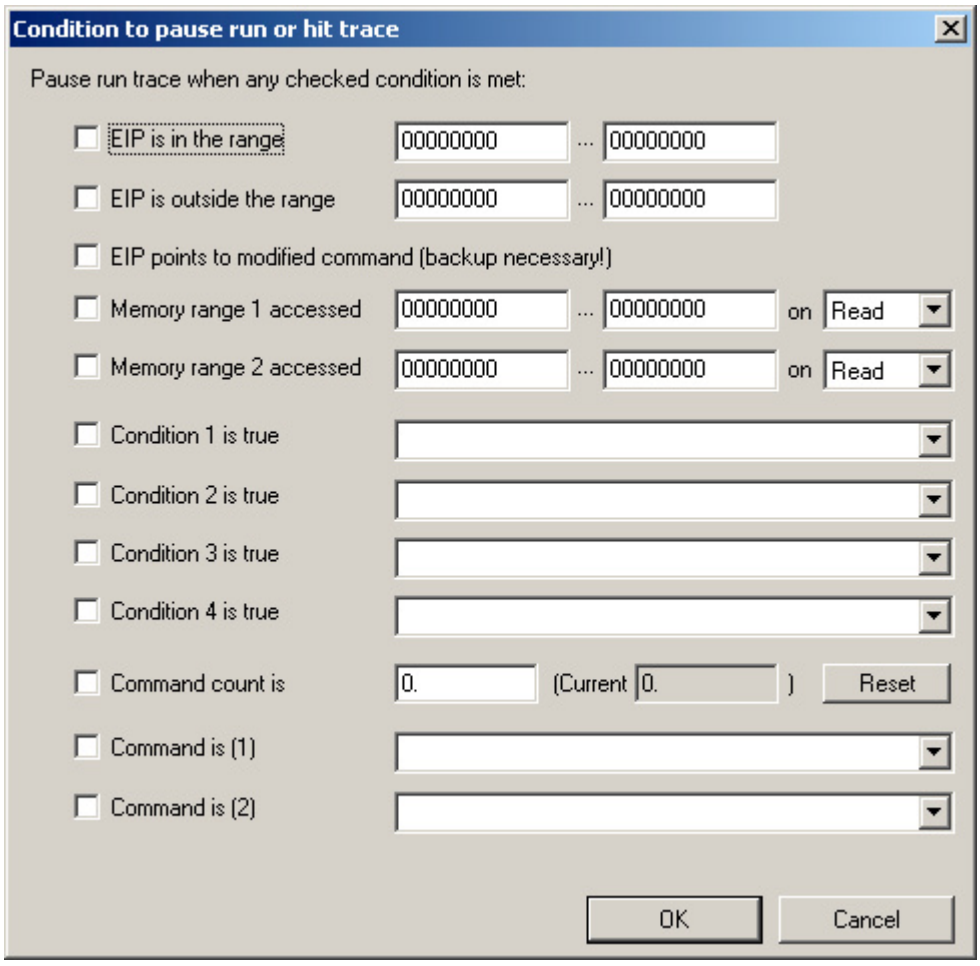
Run trace is the way to execute and protocol the debugged application command by command. In this way, one can locate the most frequently executed pieces of code, detect jumps to nowhere or just backtrace program execution that precedes some event.

In its simplest form, run trace sets breakpoint on the next command and continues execution. This method, however, is very slow. Windows need tens of microseconds to pause application and pass debugging event to the debugger. This limits run trace to 10-30 thousand commands per second on Windows XP and maybe 30-70 thousand on Windows 7.

To accelerate run trace, OllyDbg can emulate commands internally, without passing control to the debuggee. The emulation is currently limited to the 55 frequently used integer commands, like MOV, ADD, CALL or conditional jumps. If command is not known or cannot be emulated (like SYSENTER), OllyDbg passes it to the application. Still, execution speed reaches 300-600 thousand, in simple loops up to one million commands per second. In many cases, this is sufficient for the "almost real-time" behaviour of Windows applications.

Each traced command is protocolled to the large circular buffer. The protocol includes address, contents of the integer registers and flags. If you need, you may save command, FPU registes and contents of the accessed memory. Note that each option requires more memory and reduces the number of the commands that fit into the buffer. For example, if buffer is 256 MB large and all extras are turned off, it may keep up to 16.7 million commands, with extras on - only 7 to 10 million.

Most run trace options apply to the hit trace, too. Probably the most interesting trace feature is the ability to pause execution when some event occurs (**Trace | Set condition...** from the main menu):

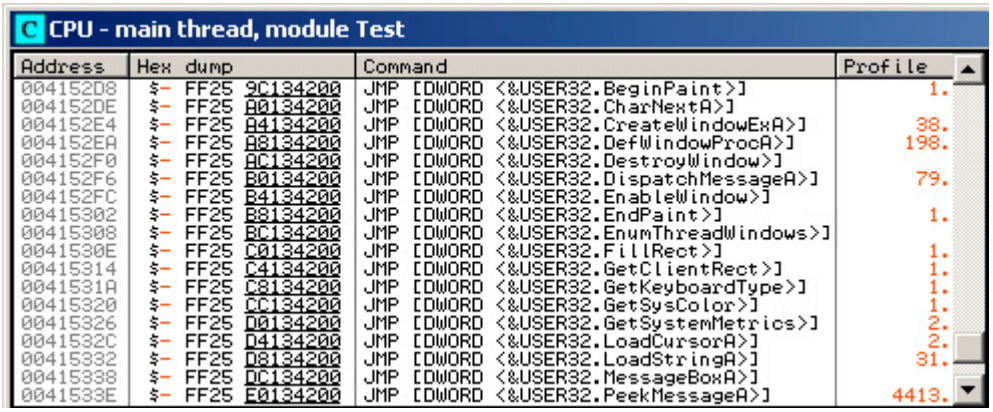


Most options are self-describing. Option **EIP points to modified command** can be used to find entry point of the program packed by self-extractor. (By the way, hit trace in this case is much faster). When you start trace and option is active, Ollydbg compares actual command code with the backup copy and pauses when they differ. Of course, backup copy must exist. The simplest way to assure it is to permanently activate option **Debugging | Auto backup user code**.

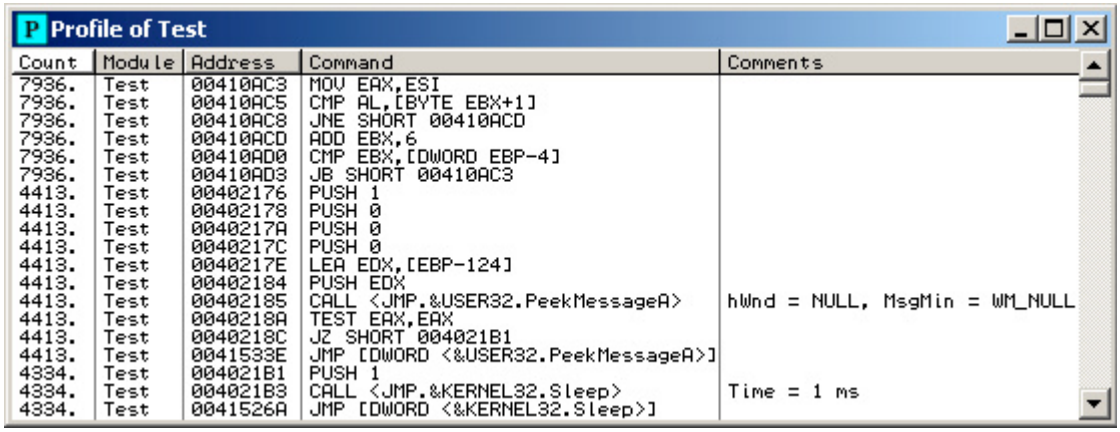
Pause on access to memory range can be implemented with memory breakpoints. However, 80x86 CPU can protect only 4096-byte chunks of memory. If memory breakpoint is set on the actively used memory block, execution will cause large number of false debugging events. To recover, OllyDbg must remove memory protection, execute single command that caused exception and restore memory breakpoint again. This requires plenty of time. If fast command emulation is active, run trace may be significantly - up to 20 times - faster than memory breakpoint.

Condition is any valid expression that evaluates to TRUE (non-zero) or FALSE (zero), for example EAX==0 or ([BYTE 450002] & 0x80)!=0. Registers are taken from the actual thread. The evaluation of conditional expressions is very quick and has only minor influence on the run trace speed.

If you need to know how frequently each traced command was executed, choose **Comments | Show profile** in the Disassembler, or **Profile selected module** or **Global profile** in the Run trace window. In the first case, Comments column will show how many times this command is present in the trace data:



Stand-alone Profile window lists traced commands, sorted by their frequency:



As you can see, snippet at address 00410AC3 was executed most frequently. It resides in the initialization routine that is automatically executed during startup. Second most frequent code, as expected, belongs to the windows loop of the main thread. Command PUSH 1 pushes parameter *Remove* of the API function *PeekMessageA()* to the stack.

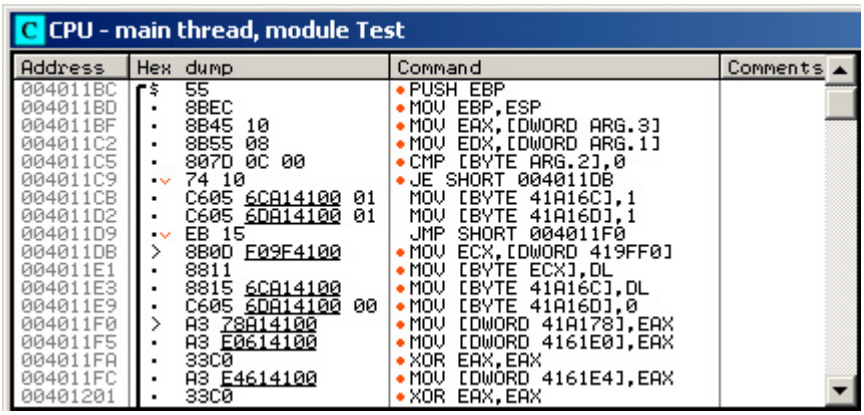
Run trace is controlled by the following options:

- **Debugging | Allow fast command emulation** - allows OllyDbg to emulate some frequently used CPU commands internally (in the contents of debugger), thus accelerating the debugging;
- **Run trace | Size of run trace buffer** - allocates memory for the circular buffer with run trace data. As a rule of thumb, one megabyte keeps 30000 - 60000 commands;
- **Run trace | Don't enter system DLLs** - requests OllyDbg to execute calls to Windows API functions at once, in the trace-over mode. Note that if API functions call user-space callbacks, they will not be traced, too;
- **Run trace | Always trace over string commands** - requests OllyDbg to trace over string commands, like REP MOVSB. If this option is deactivated, each MOVSB iteration will be protocolled separately;
- **Run trace | Remember commands** - saves copy of the traced command to the trace buffer. Only necessary if debugged application uses self-modified code;
- **Run trace | Remember memory** - saves actual contents of the addressed memory operands to the trace buffer;
- **Run trace | Remember FPU registers** - saves floating-point registers to the trace buffer;
- **Run trace | Synchronize CPU and run trace** - moves CPU selection and updates CPU registers each time you change selection in the Run trace protocol.

Hit trace

The sense of debugging is to find and remove as many bugs in the debugged application as possible. To approach this ideal, you need to execute every subroutine and every branch, otherwise latent errors are preprogrammed. But how do you know whether particular piece of code was executed? Hit trace will give you the answer.

It starts from the actual EIP. OllyDbg sets soft breakpoints on all branches that were not traced so far. After trace breakpoint is reached, it removes it and marks command as hit. This allows to check whether particular branch of code was executed. Here is an example: code at 004011CB was not yet hit:



Hit tracing is very fast. After short startup period is over, application runs with almost real-time speed. Problems may occur with indirect branches and calls, like CALL [0x405000] or JMP [0x123456+EAX*4]. In this case OllyDbg provides two options. If **Hit trace | Check destination each time** is active, OllyDbg keeps breakpoint on the command. This costs time but is safe. When **Hit trace | Use analysis data to guess destinations** is chosen, OllyDbg assumes that analysis was able to determine all possible destinations correctly and marks them all. The second way is significantly faster but may lead to missing branches, or even crashes if false branch points to data or to the middle of the command. If you are in doubt, use the first option.

Some antivirus programs place pieces of the code into the kernel memory (usually at address 0x80000000 and above). OllyDbg is unable to set soft breakpoints in the kernel memory breakpoints and continues in the step-by-step execution mode until the user memory is reached.

Other hit trace options:

- **Hit trace | Set breakpoints on known callbacks** - if active, OllyDbg sets trace breakpoint on all known callback functions when hit trace starts, so that calls from Windows API functions, like SendMessage(), can be traced;
- **Hit trace | When next destination is analysed as data: Continue hit trace / Pause hit trace** - this situation may happen either when Analyser erroneously recognized valid code as data, or if debugged program is self-modified or creates code on-the-fly. In the second case, be careful: INT3 breakpoint may have disastrous effects on the execution!
- **Hit trace | When next destination is outside the code section: Continue hit trace / Pause hit trace / Trace code command by command** - if debugged program creates code on-the fly or loads it dynamically from the disc, setting INT3 breakpoints on it may lead to crash. Step-by-step tracing is the safest, but also the slowest solution;
- **Hit trace | Keep trace between sessions** - hit trace will be saved to the .udd file and restored when you restart the application.

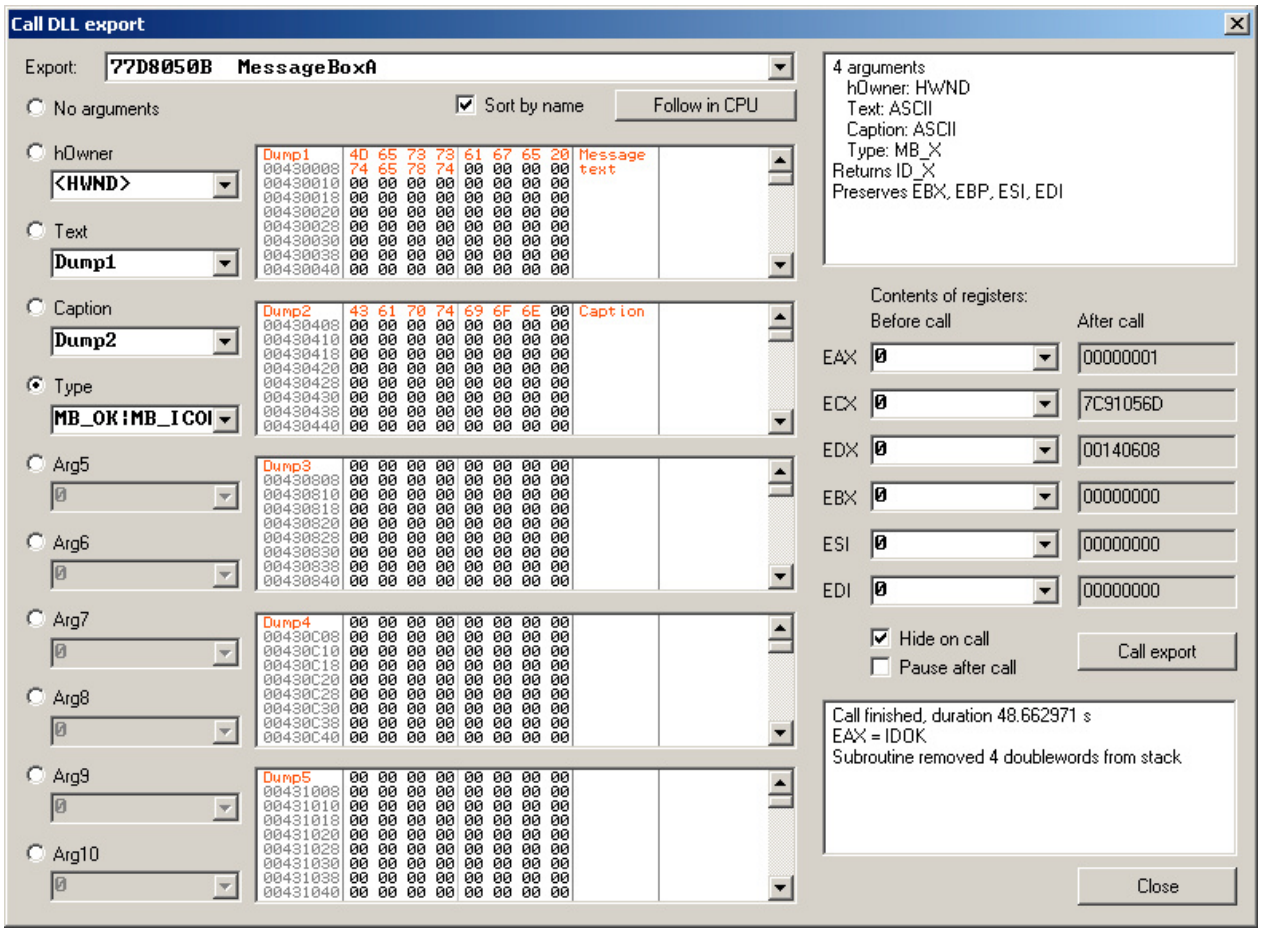
Direct DLL debugging

Loaddll.exe

OllyDbg can debug standalone dynamic link libraries (DLLs). Windows is unable to launch DLL directly, so OllyDbg uses small executable named *loaddll.exe*. This program is kept as a packed resource inside the *Ollydbg.exe*. If file you are trying to open is a dynamic link library, OllyDbg automatically extracts *loaddll.exe* and starts it, passing library name as a parameter.

Of itself, *loaddll.exe* is just a window without any controls. It loads DLL and starts infinite Windows loop, waiting for the commands from the debugger. The debugging interface is implemented in OllyDbg.

I will explain this feature on the example of Windows' API function *MessageBoxA* that resides in *USER32.DLL*. After library is loaded, you may set breakpoints and apply patches as with any other application. When you are ready, select **Debug | Call DLL export** from the main menu:



On the top there is a list of all function exported by the library. Select *MessageBoxA*. This API is in the internal database. OllyDbg knows that *MessageBoxA()* expects 4 arguments and even able to tell you their names and types. The first argument is the handle of the parent window, second ant third are pointers to the null-terminated ASCII strings, and the last one is a combination of bits with symbolic names *MB_xxx*.

As the number of arguments is known, debugger automatically checks radio button on the left that indicates the presence of 4 parameters. You may enter them in the corresponding edit controls or select predefined parameters from the drop-down list:

- **<HWND>** - handle of the window owned by the *loaddll.exe*;
- **<HINST>** - instance of *loaddll.exe*;
- **Dump1** .. **Dump 10** - pointers to 10 predefined memory areas, 1024 bytes each. Five dumps in the dialog display contents of areas **Dump1** to **Dump5**, but you may change it using **Go to | Expression** from the pop-up menu.

If function requires register parameters, enter them in the right controls. Standard Windows API functions never use registers.

Let's set the parameters. The first is the handle of the parent; here <HWND> is the best choice. Second parameter points to the ASCII string that will be displayed inside the message box. I have chosen **Dump1**, but any other memory address inside the communication area would be good, too. We want to display text "Message text". Select byte with address **Dump1**, choose **Edit | Binary edit** from the menu and in the ASCII field type "Message text".

In exactly the same way, set third parameter (**Dump2**: "Caption"). Set fourth parameter to *MB_OK*|*MB_ICONERROR*. OllyDbg knows these constants.

We are ready. Press **Call export**. Message box will appear:

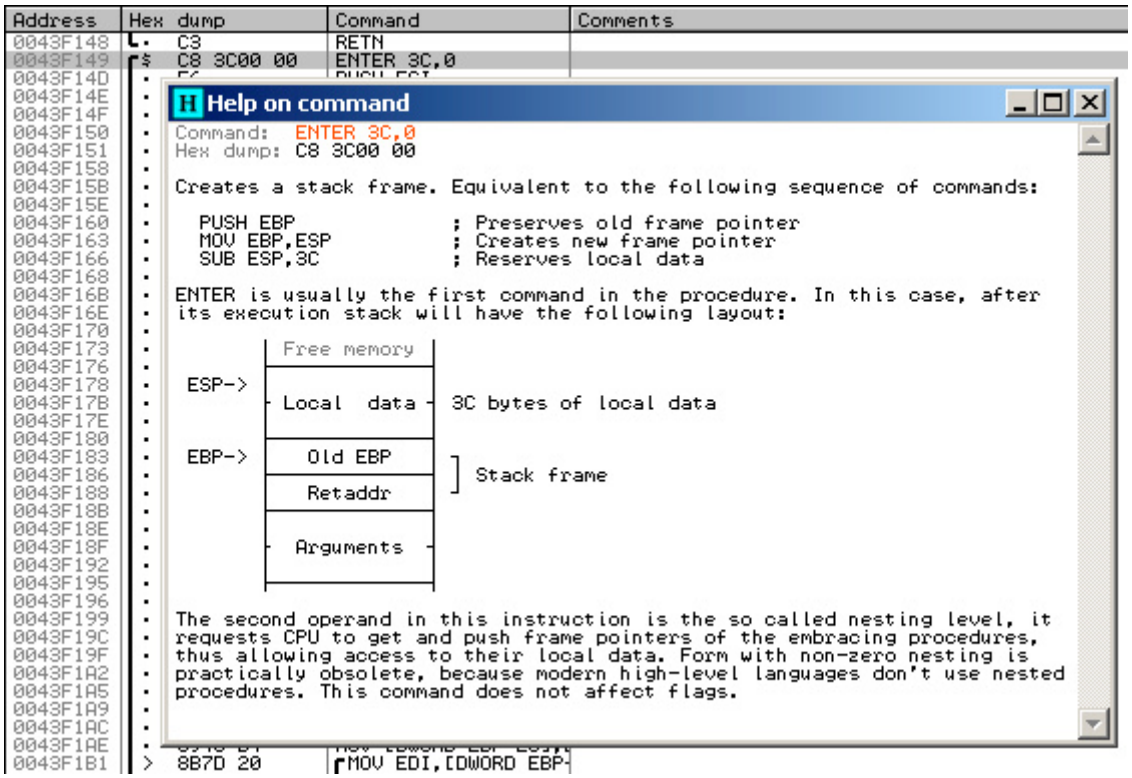
If option **Hide on call** was checked, Call DLL window will temporarily disappear. Press OK in the message box, and window will be back. Note the message in the bottom right corner: "EAX = IDOK". It indicates that *MessageBoxA()* returned *IDOK* (constant 0x1) as an answer. Another message, "Subroutine removed 4 doublewords from the stack", confirms that this Pascal-style function uses 4 stack parameters.

Loaddll.exe is written in Assembler. Its source code is freely available. There are several patch areas where you may add functionality to the program. Note that if *Loaddll.exe* is already in the OllyDbg directory, OllyDbg will use existing version.

Help

Help on commands

OllyDbg has integrated help on many 80x86 commands. Select command and choose **Help on command** from the menu (shortcut **Shift+F1**):



OllyDbg attempts to describe exactly the selected command with its operands. Currently help is available for all integer, FPU and system commands. MMX, 3DNow! and SSE commands will be added later.

Help on API functions

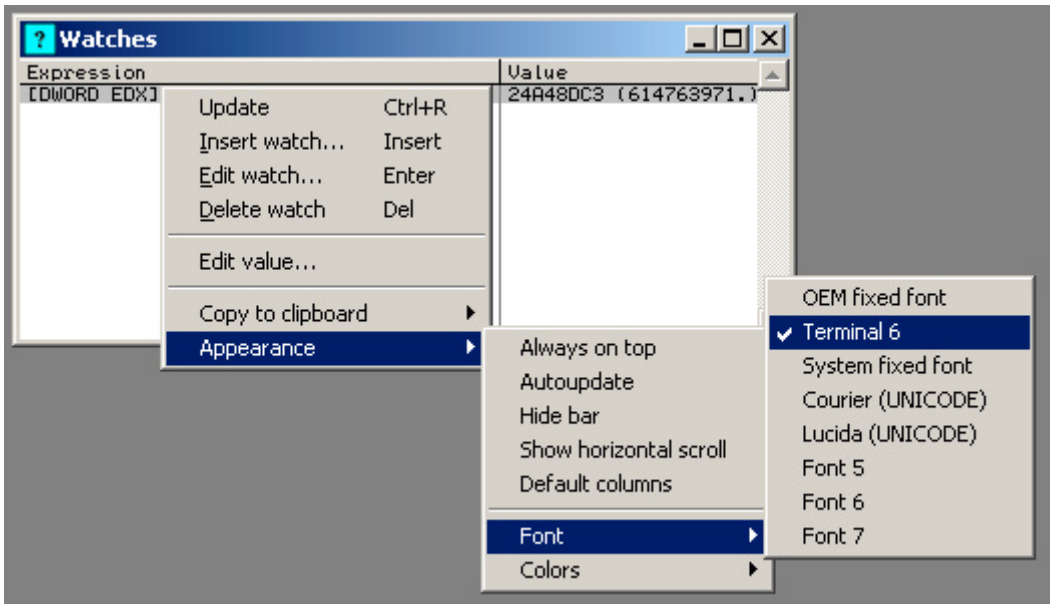
If you possess Window's API help file in *.hlp* or *.chm* format (for example, old good *win32.hlp*), you can attach it to the OllyDbg in **Options | Directories | Location of API help file**. To get help on API, select call to API function or entry point of this API and press **Ctrl+F1** (or choose **Help on API function** from the pop-up menu).

Due to the legal reasons, help file is not included into the distribution.

Customization

Fonts

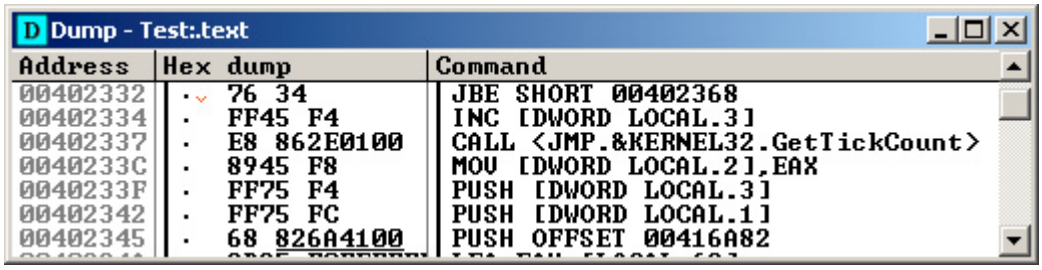
Every table window in OllyDbg gives you the possibility to select one of the 8 fonts:



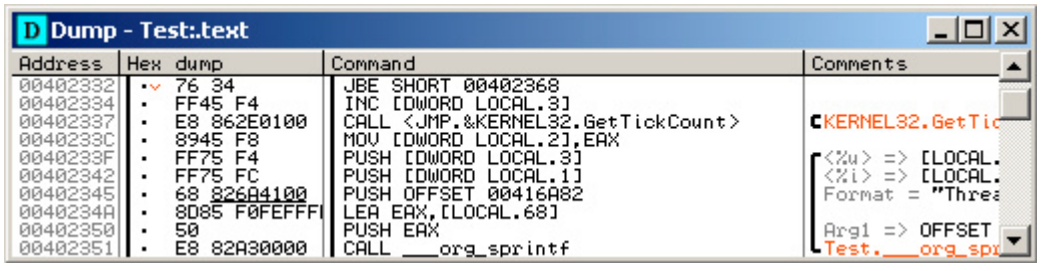
By default, OllyDbg uses font "Terminal 6". It is small and perfectly readable, therefore OllyDbg is able to display plenty of information. But Terminal 6 knows only the basic OEM character set, and may be way too small for visually impaired persons.

You may select different predefined font or redefine existing. These fonts are available when you start OllyDbg for the first time:

OEM fixed font:

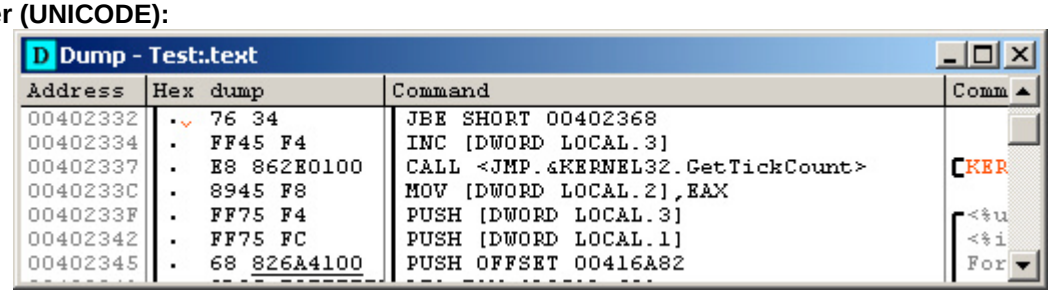


Terminal 6:

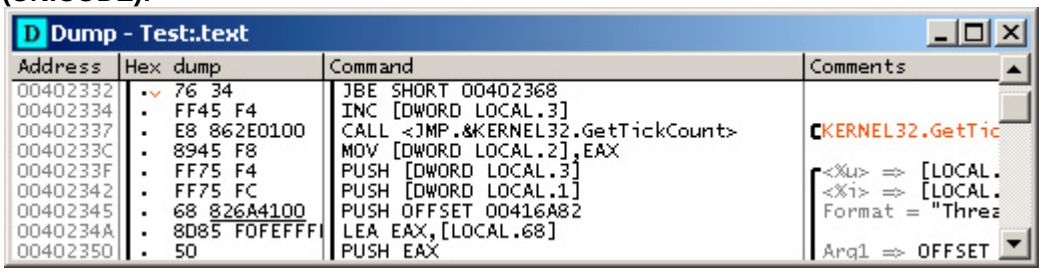


System fixed font:

Courier (UNICODE):



Lucida (UNICODE):



Fonts 5, 6 and 7 are initially identical with Terminal, System and Courier. Note that some fonts may be missing (and replaced by other fonts) on your installation of Windows.

OllyDbg allows only fixed-size fonts where all charactes have the same width. Therefore you can't use, say, Times New Roman - it simply wouldn't fit. Some characters, like W, were clipped, another, like I, would appear misadjusted.

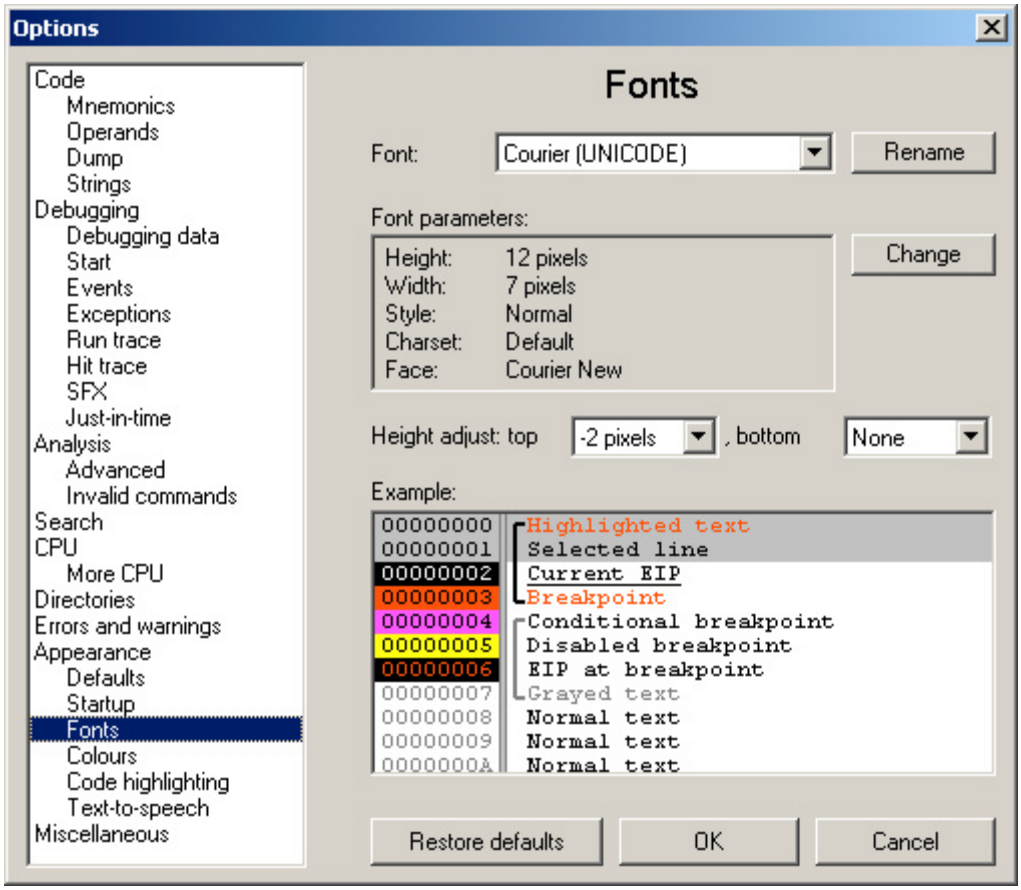
Note that fixed fonts are not as fixed-width as one may imagine. Kanji or Hangul symbols are usually twice as wide as ASCII subset. If you want to display them in the dumps without clipping, I recommend to turn on the option **Dump | Use wide characters in UNICODE & multibyte dumps**.

If you want to change font appearance, open **Fonts** options panel. Please note the following: changes that you make here apply only to the selected font. If Disassembler uses Terminal 6 and you edit OEM fixed font, you will see no changes in Disassembler pane. Either adjust Terminal 6, or select OEM fixed font in the Disassembler. Note that any modifications take effect instantly, for example when you press **OK** or **Apply** in the standard font dialog. But if you select different font at the top of the **Fonts** options panel, this does not automatically change the fonts chosen in the OllyDbg windows!

Back to the **Fonts**. Button **Rename** allows you to rename the currently selected font. This action is of rather cosmetical nature. In fact, fonts in OllyDbg windows are selected by the slot index, not by name. If Terminal 6 is selected in Disassembler and you rename Terminal 6 to Monitor 10-4, not a single pixel in the Disassembler will change, only Appearance menu will show you that currently selected font is Monitor 10-4.

Button **Change** invokes standard dialog that allows you to choose font and define its size and appearance. It lists all fixed-width fonts installed on your system.

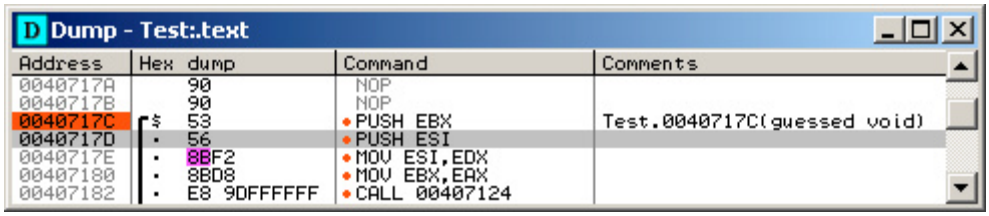
Controls **Height adjust top** and **Height adjust bottom** add or remove up to 5 pixels on the top and on the bottom of each displayed line. For example, capital letters in Terminal 6 are seven pixels high and font height, as defined in the font description, is eight pixels. One pixel distance between the lines on the screen is not sufficient, especially when fixup underlining is on. Therefore OllyDbg adds one additional pixel at the top of each line. The height of Courier New, on the other hand, is exaggerated, so it's necessary to remove two pixels on the top:



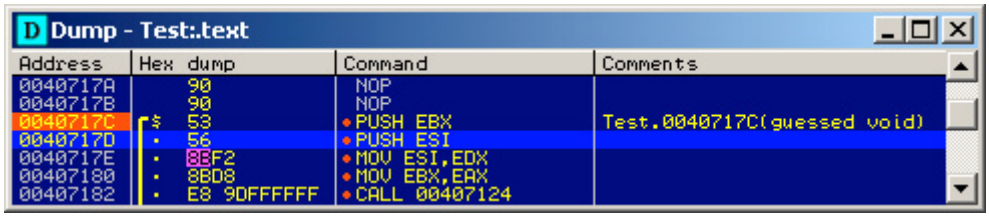
Colours

There are eight colour schemes that determine the colours used in the table windows. Each scheme can be selected into any table (pop-up menu **Appearance | Colours**). Only the first four schemes are initially defined:

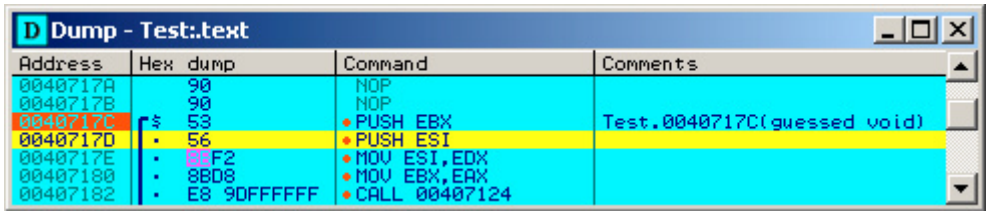
Black on white:



Yellow on blue:

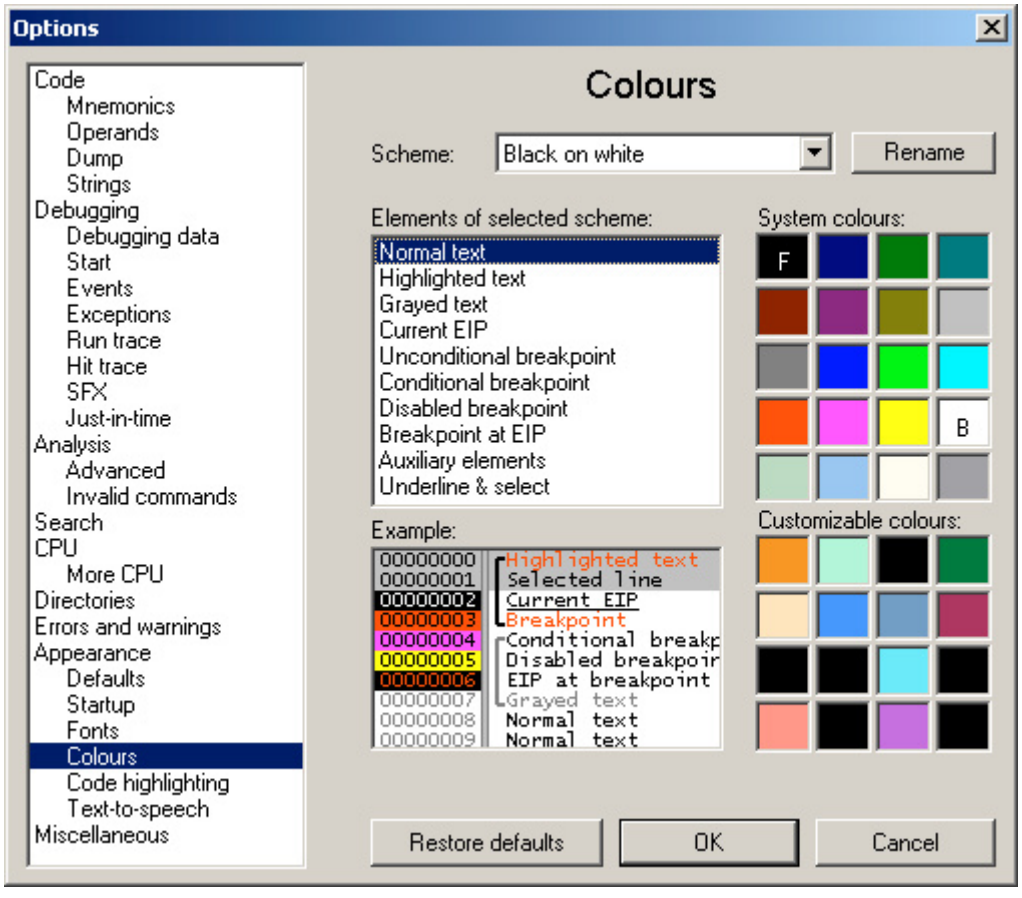


Marine:



Mostly black:

The remaining four default schemes, imaginatively named Scheme 4 to Scheme 7, repeat the first four. Colour schemes are freely editable:



Exactly as with fonts, changes that you make here apply only to the scheme that is selected on the top of the Colours panel. If scheme selected in Disassembler is Black on white and you edit Mostly black, you will see no changes in the Disassembler.

Elements have more or less descriptive names. Auxiliary elements are currently unused (but plugins may use them).

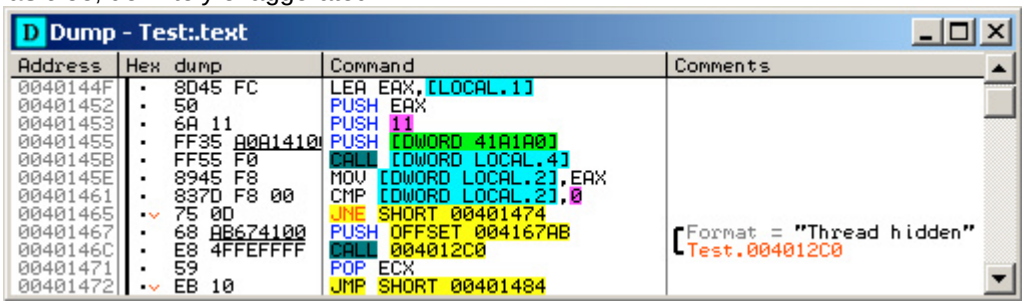
OllyDbg defines 20 basic colours which are displayed in the top half of the panel. They belong to the default colours of the 8-bit palette. This should minimize the number of palette changes in the improbable case of the ancient video card. Additionally you can define 16 custom colours - just doubleclick the corresponding square.

To change foreground, select element and click left mouse button on one square with the desired colour. Letter F moves here, indicating Foreground. Right mouse button selects background colour. The changes are applied instantly. If colour scheme in one of the visible table windows coincides with the edit, you will immediately see the results. Button **Restore default** sets colours predefined for the scheme. Button **Cancel** rolls back all modifications.

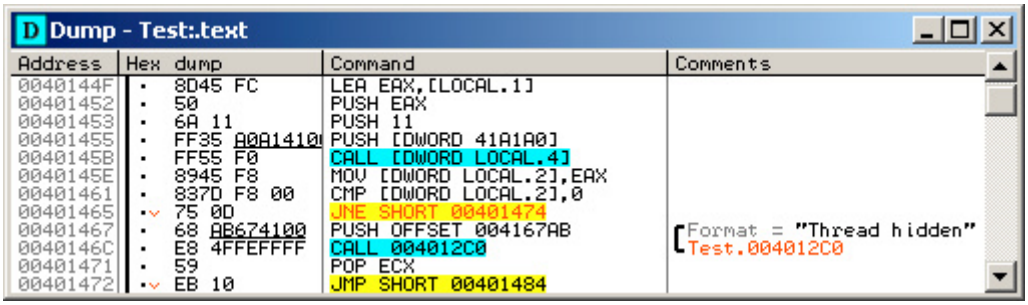
Code highlighting

There are seven possible code highlighting schemes. Initially OllyDbg defines only three:

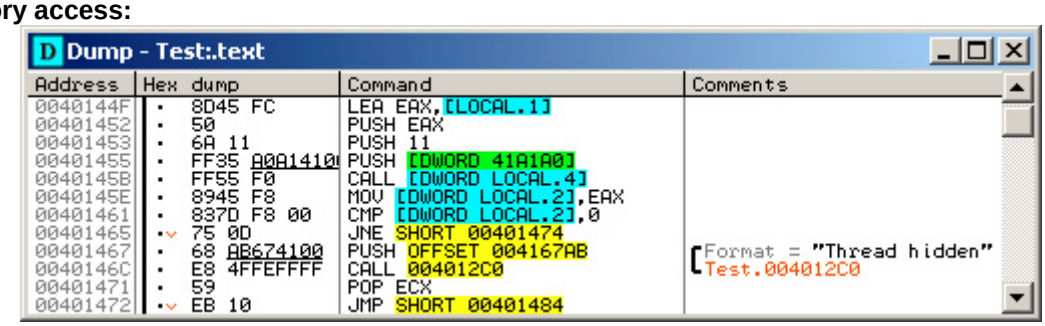
Christmas tree, definitely exaggerated:



Jumps and calls:



Memory access:



Highlightable objects are commands and optionally their operands. Commands are divided into the following groups:

Unconditional jumps	JMP, JMP FAR
Conditional jumps	JE, JNZ, ...; JCXZ
PUSH/POP	PUSH, POP, PUSHF, POPF (PUSHA and POPA are interpreted as plain commands)
Calls	CALL, CALL FAR, INT
Returns	RET, RETF, IRET
FPU, MMX, SSE	All FPU, MMX, 3DNow!, SSE and AVX commands
Bad, system and privileged commands	I/O: IN, OUT, INS, OUTS, ...; system commands: ARPL, SYSENTER, ...; privileged commands: HLT, INVLPG, LGDT,...; bad commands: LEA ESI,EDI. Note that UD2 is a documented and therefore plain command
Filling	NOPs used as a filling between the procedures: NOP, LEA ESI,[ESI], ...
Modified commands	Command that differ from backup
Plain commands	All remaining commands, like MOV or ADD

If option **Highlight operands** is active, operands are highlighted separately:

General registers	EAX, ESI, AX, AL, ...
FPU, MMX, SSE registers	ST(0), MM0, XMM0, YMM0, ...
Selectors and system registers	FS, CR0, DR0, ...
Stack memory	Memory address includes ESP or EBP
Other memory	Memory address does not include ESP or EBP
Constants pointing to memory	Constant that points to memory belonging to some module
Other constants	All remaining constants

Closely related to the code highlighting is register highlighting. If dump window shows disassembly, open context menu and choose **Highlight register | <general-purpose register>**. This option has priority over code highlighting and visualises specified register and its parts. For example if you select EAX, OllyDbg will highlight EAX, AX, AH and AL.

Shortcuts

The default shortcuts are based on the scheme used by Borland tools (I am a big fan of Borland). Most of you are probably better familiar with the Microsoft's Visual Studio. Not a problem - shortcuts can be edited on the fly. From the main menu call **Options | Edit shortcuts...** Select item and choose the combination of keys. If this combination is not allowed, reserved or conflicts with the existing shortcuts, you will be warned. If there are conflicts and you press **Apply**, conflicting shortcuts will be deleted.

Modifications to the existing set are saved to the *ollydbg.ini*. By pressing button **Save**, you can save your shorcuts to the separate file. You may freely distribute the modifications - in fact, I encourage you to do this.

GUI language

OllyDbg 2.01 supports multiple languages in user interface. Note that this is still an experimental feature.

This is how you can translate OllyDbg to the new language. Download file *ollydbg.lng*. This is a UNICODE text file:

```
EN English

// ANALYSER:206
EN Analysing %s - $ - press SPACE to interrupt

// ANALYSER:227
EN Analysis interrupted

...
```

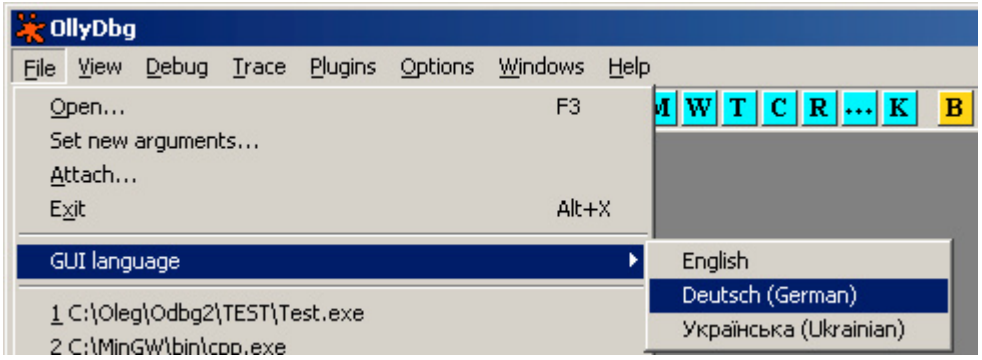
Lines // ANALYSER:206 etc. are comments that indicate the first occurance of the string in the OllyDbg sources. You may strip them from the file, but they ease the troubleshooting.

First you must define the one or more new languages. Assign them two-letter identifiers and, directly after the line "EN English", add identifiers followed by the language names, first in the new language and then in English.

Then translate, one by one, all 4100 text strings used by OllyDbg. Each translated line must begin with the language identifier and follow original line. Please take special care when translating format specifiers. Their order must remain unchanged. Don't replace format characters by the characters that look similarly but have different codes, like Latin i (U+0069) and Cyrillic i (U+0456) - this may lead to crash! Positions of ampersands in menu items must be changed to define new menu shortcuts. Dollar symbols have special meaning, keep them at a similar position:

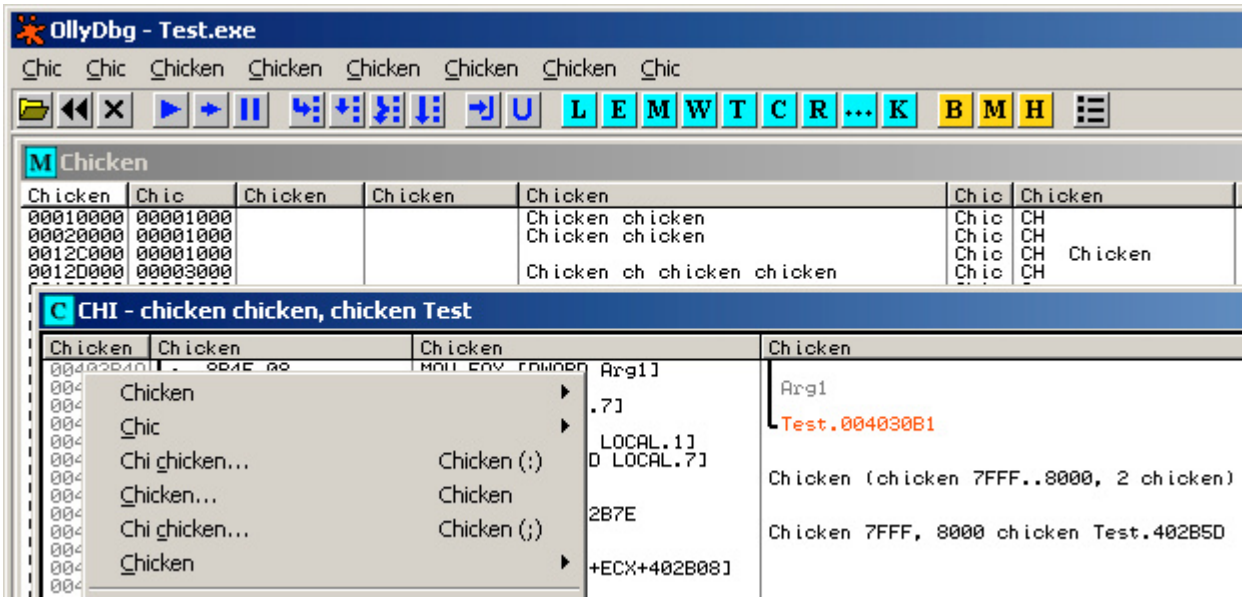
```
EN English
DE Deutsch (German)
UA Українська (Ukrainian)

// ANALYSER:206
EN Analysing %s - $ - press SPACE to interrupt
DE Analysiere %s - $ - zum Anhalten drücken Sie die Leertaste
UA Аналізую %s - $ - щоб зупинити, натисніть пропуск

// ANALYSER:227
EN Analysis interrupted
DE Analyse gestoppt
UA Аналіз перервано

...
```

When you are ready, save this file to the directory containing *ollydbg.exe* and start OllyDbg. To change language, open main menu and choose new language from the **File | GUI language**. This menu is always in English:



Language will change instantly. It is not necessary to restart the debugger. Only few elements, like window titles, will be translated first when you reopen the window. Translatable commens created by the Analyser are kept in the *.udd* file; to change them, re-run analysis.

To debug this feature, I have translated all strings to chicken. A good explanation is available here: www.youtube.com/watch?v=yL_-1d9OSdk and as PDF here: isotropic.org/papers/chicken.pdf. You can download chicken language file from the OllyDbg homepage:



Renaming the ollydbg.exe

If you rename main OllyDbg executable file to, say, *alias.exe*, it will automatically change many settings: name of the main window (**alias**), name of the initialization file where it keeps the settings (*alias.ini*), class names of the windows (alias_ODWin, alias_ODMDI, ...) etc. However, there is a problem. Plugins are all statically linked to the file *ollydbg.exe*. They cannot link to the file with different name.

To overcome this problem, OllyDbg attempts to temporarily change the name of the main module in the memory. This works well under Windows XP. However, I can't guarantee that this will work under any other version of Windows.

Apologies

"We apologize for inconveniences"
Douglas Adams

I'm not a native English speaker. Please forgive me all the grammatical errors. I would be very pleased if you let me know about especially unhappy phrases and suggest replacement.

I apologize also for the semantical errors in the C code. They usually result in a window reporting that processor exceptionally dislikes command or data at some address. Of course, I can only blame my otherwise excellent compiler because it did literally what I wrote, not what I meant. Please forgive him... and send me the file errorlog.txt that will be created on this event:

```
OLLYDBG EXCEPTION PROTOCOL

This file is created by OllyDbg due to unrecoverable error. It
contains data necessary to locate and remove this and previous
errors. Please describe circumstances that preceded exception:

>
>
>
>

and email protocol to:

   Ollydbg@t-online.de

Feel free to remove any private data. Thank you for your help!

Operating system:   5.1.2600, platform 2 (Service Pack 2)
OllyDbg version:    2.01.00 alpha 4
Exception code:     C0000094
Exception address:  022B12B7
Executable module:  C:\Oleg\Odbg2\plugins\Bookmark.dll

EAX=00000001  EBX=0012EA2C  ECX=00000000  EDX=00000000
ESP=0012E7CC  EBP=0012E7D0  ESI=0012F61C  EDI=00564858
EIP=022B12B7  EFL=00210206

Code dump:
022B1277  C3 55 8B EC  83 7D 0C 01  75 08 8B 45  08 A3 A0 B6
022B1287  2B 02 B8 01  00 00 00 5D  C2 0C 00 90  90 55 8B EC
...
```

I apologize also for the inconveniences. If you miss some very useful function, please send me a mail, but don't expect too much.

That's all for now!

- Oleh Yuschuk, also known as Olly