OllyDbg 2.0 Brief Help

Table of contents

© squashed bug 2.0

Introduction

This guide is not a detailed description of the OllyDbg 2.0. Rather, it highlights some interesting and non-obvious features of the new debugger.

Differences between versions 1.10 and 2.00

The second version of the 32-bit debugger OllyDbg is redesigned practically from scratch. As a result, it is faster, more powerful and much more reliable than its predecessor. Well, at least in the future, because many useful features from the version 1.10 are not yet available in 2.00.

However, version 2.00 already contains many new features that were not available before. Among them:

- Full support for SSE instructions, including SSE3 and SSE4. 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;
- Built-in help for integer and FPU commands;
- Option to pause on TLS callback;
- Option to pass unprocessed exceptions to the unhandled exception filter.

OllyDbg 2.0 overview

OllyDbg 2.0 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 2000 and Windows XP, but should work under any 32-bit Windows version: 95, 98, ME, NT, 2000, XP, 2003 Server, Vista, Windows 7 and so on. Version 2.00 will not work under 64-bit Windows yet! For a comfortable debugging you will need at least 500-MHz processor. OllyDbg is memory hungry. If you debug large application with all features enabled, it may allocate 200-300 megabytes for backup and analysis data.

Supported instruction sets. OllyDbg 2.0 supports all existing 80x86-compatible CPUs: MMX, 3DNow!, including Athlon extensions, and SSE instructions up to SSSE3 and SSE4.

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, 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, PEB, Thread data blocks and so on.

Help. OllyDbg 2.0 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, 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 7800) constants, like window messages, error codes or bit fields, and decodes them in calls to known functions.

Known functions. OllyDbg recognizes by name more than 2200 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. Notice however that they can be remnants from the previous calls. 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.

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 and so on. If multiple locations are found, you can quickly navigate between them.

Breakpoints. OllyDbg supports all common kinds of breakpoints: INT3, memory and hardware. You may specify number of passes and set conditions for pause. Breakpoints may conditionally protocol data to the log. Number of INT3 and memory breakpoints is unlimited: in the extreme case of hit trace, OllyDbg

sometimes sets more than 100000 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 INT3 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), 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.

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

And much more! This list is far from complete, there are many features that make OllyDbg 2.0 the friendly debugger.

(No) registration

OllyDbg 2.00 is Copyright (C) 2000-2010 Oleh Yuschuk. 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 2.00.

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 2.00 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 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.

The Software is owned by Oleh Yuschuk ("Author") and is Copyright (c) 2000-2010 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.

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 2.00 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 - 2.00 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 act as a network client or server, or send any data to any other computer by any means (except for remote files specified by user), or act as a Trojan Horse of any kind;

- 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 shareware, nor any features that limit the functionality of OllyDbg 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 *Odbg200.zip* to this folder. If necessary, drag-and-drop *ollydbg.exe* to the desktop to create shortcut.

If you are a hardcore user and run OllyDbg on Windows NT 4.0, you will need *psapi.dll*. This library is not included.

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 hardly possible. Please upgrade your OS. Anyway, I will discard support for Windows 95, 98 and ME in the future.

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 some 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, I plan to release Disassembler 2.0 under GPL v3.

Description of .udd file format is available on request. It is free.

Assembler and disassembler

General information

OllyDbg 2.0 supports all 80x86 commands, FPU, MMX, 3DNow!, SSE, SSE2, SSE3, SSSE3 and SSE4 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's 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.;
- No-operand forms of binary FPU commands are not supported, use two-operand forms (FADDP ST(1),ST instead of FADDP);
- 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;
- SMSW: register is decoded as 16-bit (only low 16 bits are used anyway);
- 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, however, uses only implicit no-operand form.

Disassembling options
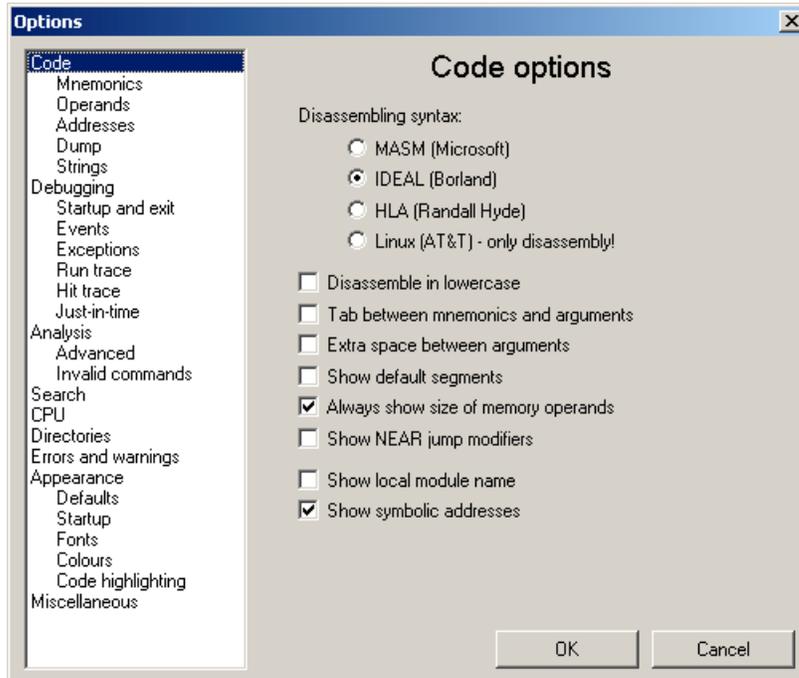
OllyDbg supports four different decoding modes: MASM, Ideal, HLA and AT&T.

MASM is the de facto standard of the assembler programming. Ideal mode, introduced by Borland, is very similar to MASM but differently decodes memory addresses. 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. AT&T syntax is popular among the Linux programmers.

Example of MASM syntax:

Example of Ideal syntax:

Example of HLA syntax:

Example of AT&T syntax:

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

HLA is a public domain software, you can download it together with documentation and sources from http://webster.cs.ucr.edu.

Other code display options are more or less self-explanatory.

Undocumented 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.

Caveats

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

    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}.

Analyser

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.
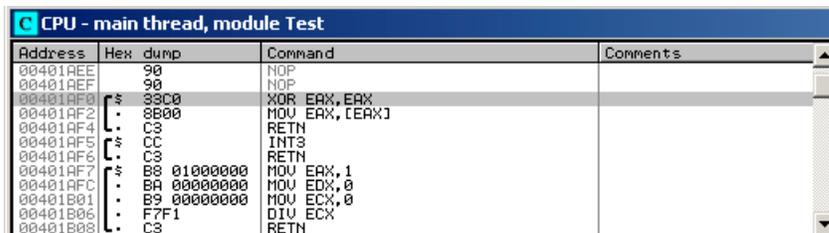
How is it possible at all? OllyDbg makes 12 passes through the program, each time gathering information that will be used on the next steps. For example, on the first pass it disassembles every possible address within the code sections and checks whether this can be the first byte of the command. Valid 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, many 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 their number 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 long fat parenthesis in the dump column. Dollar sign (**$**) to the right marks call destinations, sign "greater than" (**>)** - jump destinations, point (•) – other analysed commands. Three procedures on the picture below implement writing to the memory at address 0 (NULL pointer), INT3 and division by zero in the test application:

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:

```
                (unused memory)
    ESP->   local N             [LOCAL.N]
            ...                  ...
            local 2             [LOCAL.2]
            local 1             [LOCAL.1]
    EBP->   Old EBP             [LOCAL.0]
            Return address      [RETADDR]
            argument1           [ARG.1]
            argument2           [ARG.2]
            argument3           [ARG.3]
            ...
```

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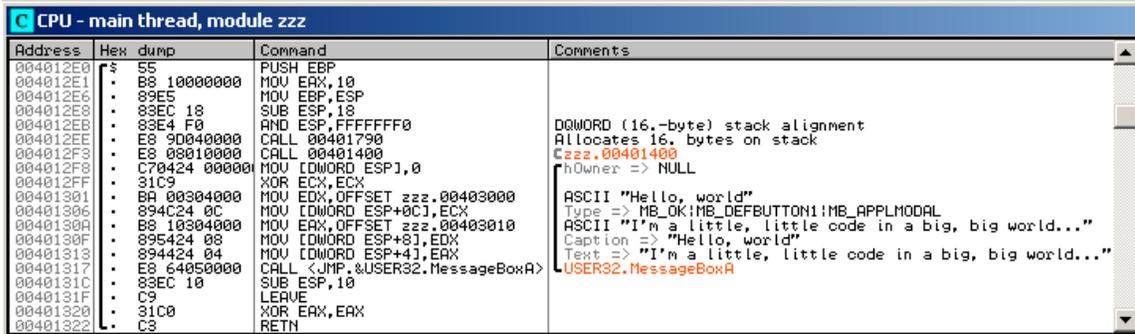
Some compilers do not push each argument separately. Instead, they allocate memory on the stack (SUB ESP,nnn) and then write arguments using ESP as index. First doubleword argument is [ESP], second - [ESP+4] and so on.

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 2 was able to recognize the arguments.

Known functions

OllyDbg contains descriptions of more than 2200 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 7800 symbolic constants, grouped into 490 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.

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, together with the C code. Selected command is loop exit. Long red arrow shows its destination:

```
   ...
   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;
```


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.

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) {...}
```

Prediction of registers

Look at this code snippet:

Analyser 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 arguments are remove from the stack by the calling program. Command ADD ESP,8 after the call does exactly this: it removes 8 bytes of data, or two doublewords from the stack. 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 0040960B just adds 0x1D37 to ESI. But where ESI is defined?

If you look through the procedure, you will be able to restore the structure of the code:

```
...
MOV ESI,OFFSET Ot.0045E7EC ; Here ESI is defined
...
PUSH ESI
...                        ; Inlined strcpy()
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 modification by inlined *strcpy()*: it uses MOVS. But then you see two calls to unknown functions xxx and yyy. Why analyser is sure that they leave ESI unchanged? Well, either analyser determined this directly from the code of xxx and yyy, or you told it by selecting corresponding analysis option. All API function and many separately compiled procedures use *stdcall* convention: functions should preserve registers EBX, EBP, ESI and EDI.

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 using them as a single data block.

OllyDbg treats application's memory as a set of independent blocks. 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 will be unable to see the whole code at once.

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.

Kernel memory

Kernel (system) memory, allocated in the high memory area, is not directly accessible to the application. However, OS or driver may grant access to portions of this memory. For example, some antivirus programs redirect *LoadLibrary*, *GetProcAddress* and similar API functions to the kernel memory. OllyDbg can neither modify kernel memory, nor change memory attributes, nor set breakpoints of any kind. Redirected API calls are traced in a single-step trap mode.

Ring 3 application has no means to get layout of the kernel memory, and direct scan can be very slow. For this reason, OllyDbg displays kernel memory as a single block where unavailable pages are filled with zeros instead of question marks, as in all other cases. The only available operation on kernel memory is search. Attention, full search may take several minutes!

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, modifications are highlighted. 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 CPU window. Backup is necessary if you request run trace pause on modified command. See detailed description below in the section "Run trace and profiling".

Break on memory access

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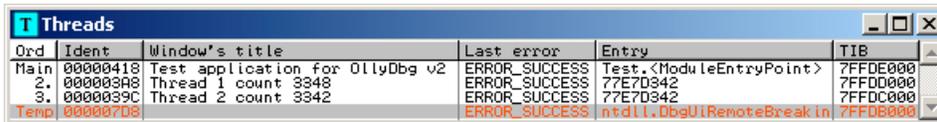
This kind of memory breakpoint is useful if you need to find calls to DLL or returns from such calls. Just place breakpoint on the code section.

Threads

General information

OllyDbg can debug multithreaded applications. Main thread of the Debuggee has ordinal number 1. Subsequent thread created by application 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.

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

Column 'Window's title' 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' contains the most recent error code returned by call to API function. This is the value returned by *GetLastError()*.

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

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 signal is paused.

In this case pause program and run it in the all-threads mode. Temporary breakpoint remains active and will be hit shortly.

Debugging

Attaching to 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.

While attaching, Windows creates new thread in the contents of the Debuggee. This thread executes DbgBreakPoint(), thus giving Debugger the chance to make all necessary preparations. The thread is marked as temporary by the OllyDbg. Under Windows 2000, OllyDbg is unable to recognize this thread as temporary and reports it as an ordinary thread.

Attaching to running process is controlled by the following options:
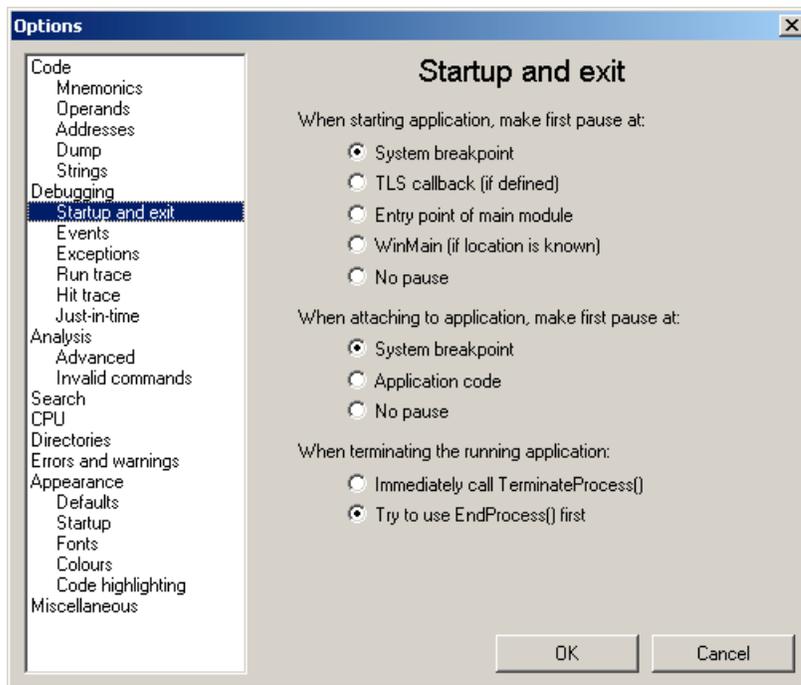
When attaching to application, make first pause at: System breakpoint - asks OllyDbg to pause application on the system breakpoint in the temporary thread.
When attaching to application, make first pause at: Application code - asks to pause application in the main thread at the location that was executing at the moment of attaching. Usually this is *ntdll.dll*.
When attaching to application, make first pause at: No pause - application should continue execution as soon as possible.

Detaching from the debugging process

Under Windows XP and higher OS versions, you can detach from the running process. From the main menu, select **File | Detach**. OllyDbg automatically removes all breakpoints, but does not resume suspended threads. Don't forget to do it manually!

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**.

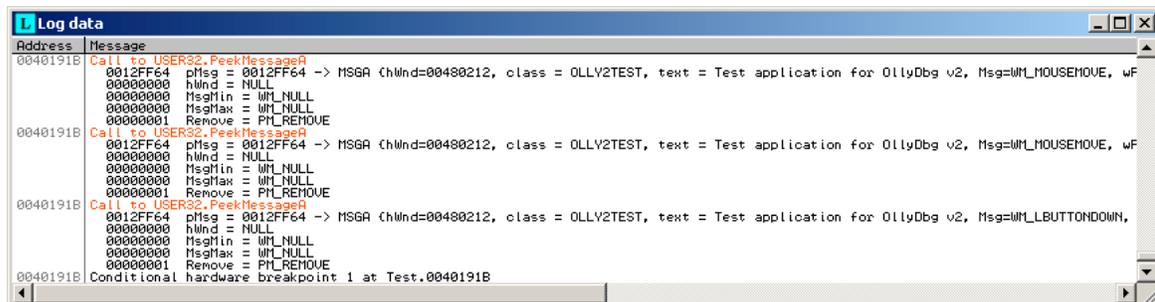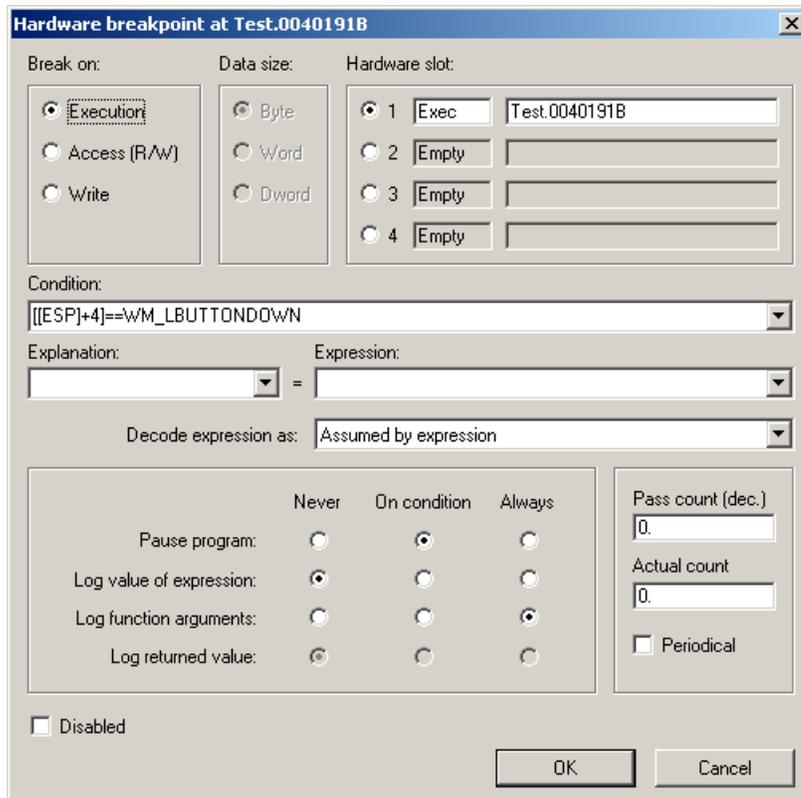# Breakpoints

OllyDbg supports three types of breakpoints:

- INT3 breakpoints on code execution;
- Memory breakpoints on memory access, write to memory or code execution;
- Hardware breakpoints on memory access, write to memory or code execution.

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

Number of INT3 and memory breakpoints is unlimited. 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 chapter "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.

Number of hardware breakpoints is limited to 4 by the CPU.

Simple breakpoint pauses execution each time the command is executed or memory is accessed. You may also set conditional logging breakpoints with many additional options. The picture below represents hardware breakpoint on the call to PeekMessageA() at address 0x0040191B in the main loop of the test application and corresponding protocol. Each time the breakpoint is hit, the parameters of the call are saved to the log. We want to pause application when received message is WM_LBUTTONDOWN. First parameter in the call is the pointer to the structure MSGA. It is pushed last and therefore has address [DWORD ESP], or simply [ESP]. Message identifier is the second element of the structure, doubleword at address [ESP]+4. Its contents, [[ESP]+4] is compared with the constant WM_LBUTTONDDOWN. If they are equal, execution will be paused:

Hardware breakpoint at Test.0040191B

Break on:
- ● Execution
- ○ Access (R/W)
- ○ Write

Data size:
- ○ Byte
- ○ Word
- ○ Dword

Hardware slot:
- ● 1 Exec Test.0040191B
- ○ 2 Empty
- ○ 3 Empty
- ○ 4 Empty

Condition:
[[ESP]+4]==WM_LBUTTONDOWN

Explanation: [   ] = Expression: [   ]

Decode expression as: Assumed by expression

	Never	On condition	Always
Pause program:	○	●	○
Log value of expression:	●	○	○
Log function arguments:	○	○	●
Log returned value:	●	○	○

Pass count (dec.)
0.

Actual count
0.

☐ Periodical

☐ Disabled

[OK] [Cancel]

D Structure MSGA at 0012FF64

Address	Hex dump	Decoded data	Comments
0012FF64	12024800	DD 00480212	hWnd = 00480212, class = OLLY2TEST, text = Test application for OllyI
0012FF68	01020000	DD 00000201	Msg = WM_LBUTTONDOWN
0012FF6C	01000000	DD 00000001	wParam = 1
0012FF70	FC006600	DD 006600FC	lParam = 6600FC
0012FF74	09C32B03	DD 032BC309	time = 53199625.
0012FF78	2B010000	DD 0000012B	pt_X = 299.
0012FF7C	A8000000	DD 000000A8	pt_Y = 168.

L Log data

Address	Message
0040191B	Call to USER32.PeekMessageA
	0012FF64 pMsg = 0012FF64 -> MSGA (hWnd=00480212, class = OLLY2TEST, text = Test application for OllyDbg v2, Msg=WM_MOUSEMOVE, wF
	00000000 hWnd = NULL
	00000000 MsgMin = WM_NULL
	00000000 MsgMax = WM_NULL
	00000001 Remove = PM_REMOVE
0040191B	Call to USER32.PeekMessageA
	0012FF64 pMsg = 0012FF64 -> MSGA (hWnd=00480212, class = OLLY2TEST, text = Test application for OllyDbg v2, Msg=WM_MOUSEMOVE, wF
	00000000 hWnd = NULL
	00000000 MsgMin = WM_NULL
	00000000 MsgMax = WM_NULL
	00000001 Remove = PM_REMOVE
0040191B	Call to USER32.PeekMessageA
	0012FF64 pMsg = 0012FF64 -> MSGA (hWnd=00480212, class = OLLY2TEST, text = Test application for OllyDbg v2, Msg=WM_LBUTTONDOWN,
	00000000 hWnd = NULL
	00000000 MsgMin = WM_NULL
	00000000 MsgMax = WM_NULL
	00000001 Remove = PM_REMOVE
0040191B	Conditional hardware breakpoint 1 at Test.0040191B

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 up to hundred microseconds to pause application and pass debugging event to debugger. This limits run trace to 10-30 thousand commands per second.

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 code, FPU registes and contents of the accessed memory. Note however that each additional option reduces the number of the commands in the buffer. For example, if buffer is 256 MB large and all extras are turned off, it keeps up to 16.7 million commands, and with extras on - only 7 to 10 million.

Probably the most interesting feature of the run trace is its ability to pause execution when some event occurs (command Trace|Set condition... from the main menu):
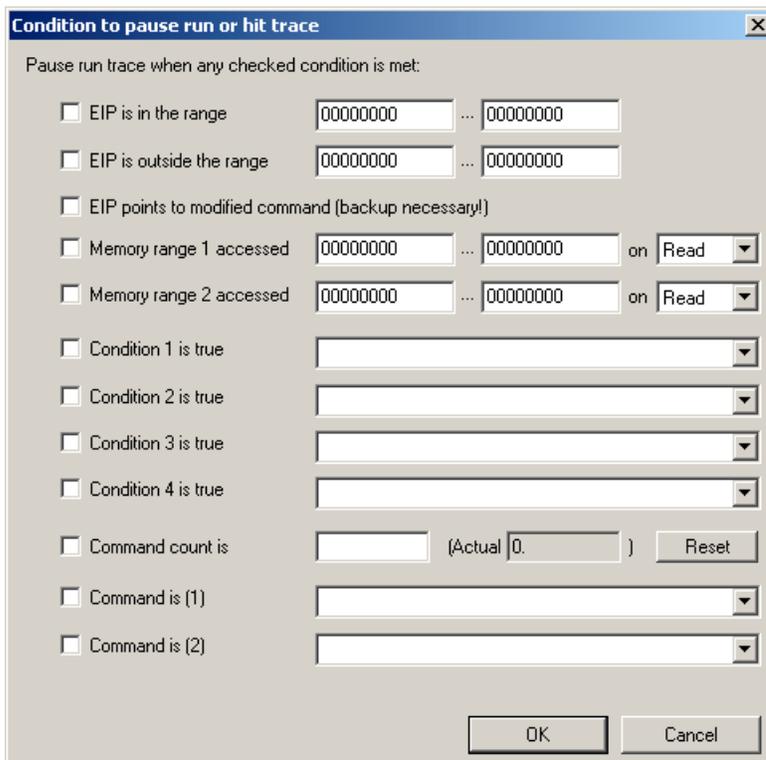
Option "EIP points to modified command" can be used to find entry point of the program packed by self-extractor. (By the way, hit trace in this case is much faster). When you start run or hit trace and option is active, Ollydbg compares actual command code with the backup copy and pauses when they differ. Of course, backup copy must exist. The simplest way to assure it is to activate **Options|Debugging options|Auto backup user code**.

Pause on access to memory range can be implemented with memory breakpoints. Note however that 80x86 CPU protects memory only in 4096-byte chunks. If memory breakpoint is set on the small part of 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 command emulation is activated, run trace may be significantly - up to 20 times - faster than memory breakpoint.

Condition is any valid expression, 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. In the first case, Comments column will show how many times this command is present in the trace data:
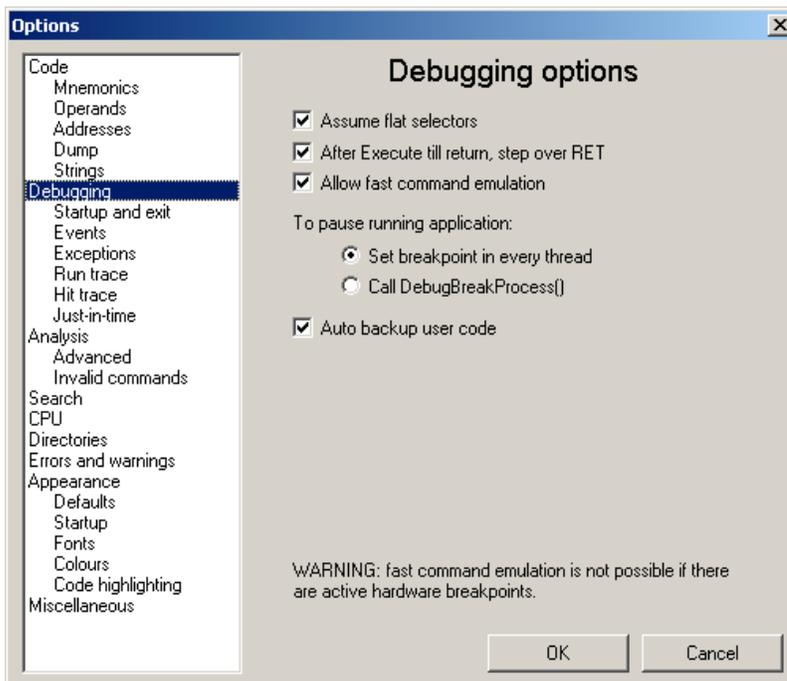

In the two remaining cases Profile window will list traced commands, sorted by their frequency:

As you can see, snippet at address 0040FF93 in module *Test* was executed most frequently.

Run trace is controlled by the following set of options:

Allow fast command emulation - allows OllyDbg to emulate some frequently used CPU commands internally, thus accelerating the debugging. Emulation is currently not compatible with the hardware

breakpoints. If some hardware breakpoint is active, emulation is disabled;
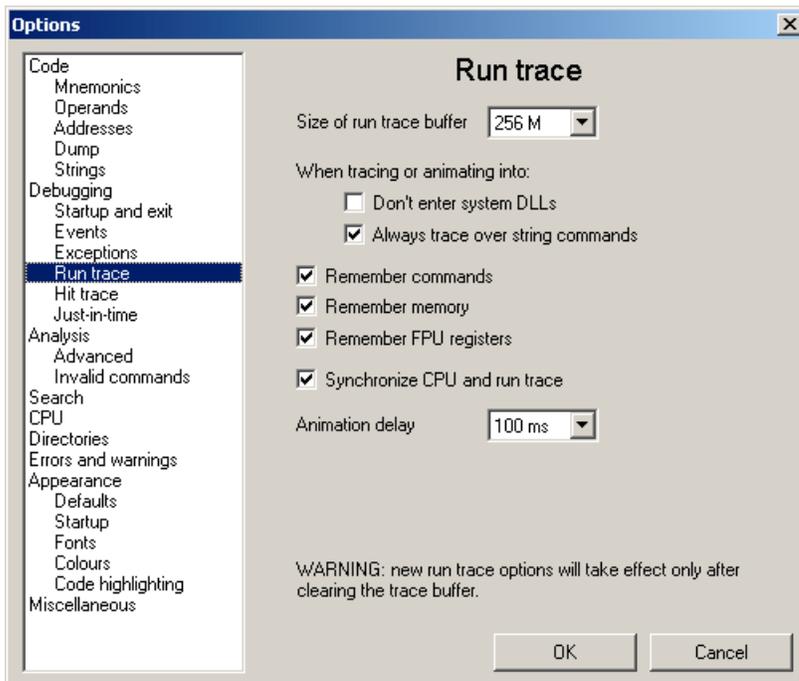
Size of run trace buffer - allocates memory for the circular buffer with run trace data. As a rule of thumb, one megabyte keeps 30000 - 60000 commands;

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;

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;

Remember commands - saves copy of the traced command to the trace buffer. Only necessary if debugged application uses self-modified code;

Remember memory - saves actual contents of the addressed memory operands to the trace buffer;

Remember FPU registers - saves floating-point registers to the trace buffer;

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 all bugs from the debugged application. 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 location. OllyDbg sets INT3 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 at a glance whether some branch of code was executed. Here is an example: code at 004011CB was never executed.

```
C CPU - main thread, module Test
Address  | Hex dump     | Command
004011BC | ┌$  55       | ◆PUSH EBP
004011BD | │.  8BEC     | ◆MOV EBP,ESP
004011BF | │.  8B45 10  | ◆MOV EAX,[DWORD ARG.3]
004011C2 | │.  8B55 08  | ◆MOV EDX,[DWORD ARG.1]
004011C5 | │.  807D 0C  | ◆CMP [BYTE ARG.2],0
004011C9 | │.✓ 74 10    | ◆JE SHORT 004011DB
004011CB | │.  C605 8C8 | MOV [BYTE 418B8C],1
004011D2 | │.  C605 8D8 | MOV [BYTE 418B8D],1
004011D9 | │.✓ EB 15    | JMP SHORT 004011F0
004011DB | │>  8B0D 108 | ◆MOV ECX,[DWORD 418A10]
004011E1 | │.  8811     | ◆MOV [BYTE ECX],DL
```

Hit tracing is astoundingly fast, after short startup period application runs with almost full speed. Problems may occur with indirect branches and calls, like CALL [0x405000] or JMP [0x123456+EAX*4]. In this case you have two options: either to check the destination each time, or use analysis results to guess the set of possible destinations. The first way is more reliable, the second is significantly faster. If you are in doubt, use the first option.

Some malware and antivirus programs place the pieces of the code in the kernel memory (usually 0x80000000 and above). OllyDbg is unable to set INT3 breakpoints and continues in the step-by-step execution mode until the user memory is reached.

Hit trace is controlled by the following set of options:
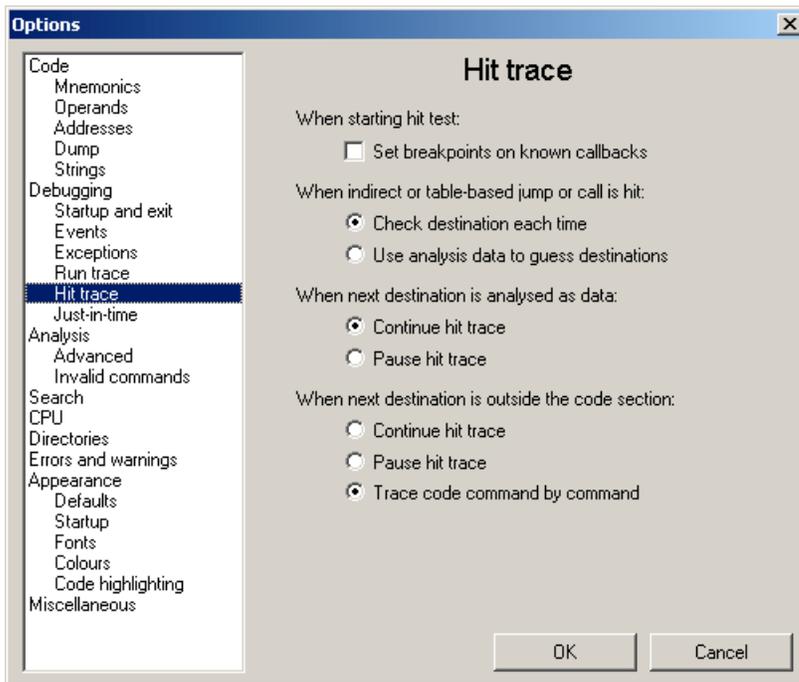
Set breakpoints on known callbacks - if active, OllyDbg sets trace breakpoint on all known callback functions when hit trace starts, so that calls from Windows API functions, like SendMessage(), can be traced;

When indirect or table-based jump or call is hit: Check destination each time / Use analysis data to guess destinations - controls tracing of indirect jumps and calls. The first option assures reliable hit trace, the second makes it faster;

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!

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 option.

Help

Help on commands

OllyDbg has integrated help on 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 in the next release.

Help on API functions

If you possess old Window's API help file *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 system DLL and press Ctrl+F1 (or choose **Help on API function** from menu).

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

Search

Imprecise Assembler search patterns

When searching for command or sequence of commands, you can specify imprecise Assembler patterns that match many different instructions. 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 keywords:

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 the following 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 the following code:

```
MOV EAX,[location 1]
CMP EAX,[location 2]
```

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

For example, 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]
```

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].

Editing

Binary edit

You have many ways in which you can edit code and data. The simplest is the binary edit, where you can edit your precious bytes as ASCII text, UNICODE text or directly in hexadecimal format. Select the piece of code or data and choose **Edit | Binary edit** from the pop-up menu:
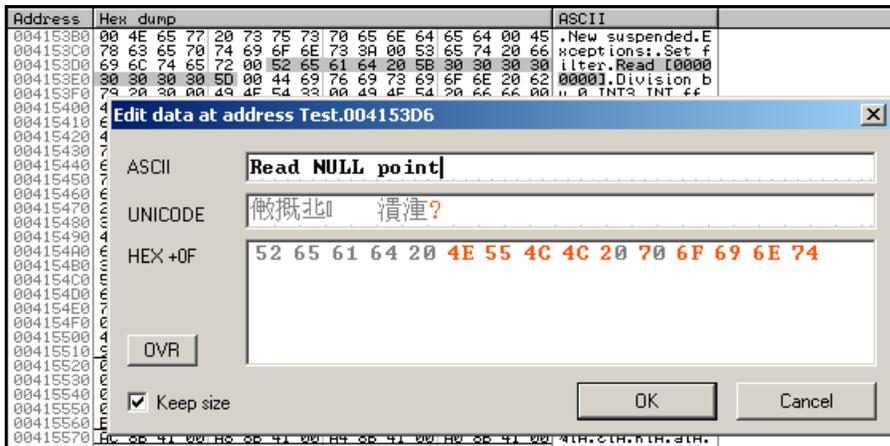
Data and modifications are displayed simultaneously in all supported formats. Note that default UNICODE font is Courier, which in most cases does not contain Far East characters. If you need them, please define new font (**Options | Fonts**) and select it as a UNICODE edit font in **Options | Appearance**.

Small button on the left toggles between insert (**INS**) and overwrite (**OVR**) modes. Be careful, hexadecimal editor in insert mode shifts data in 4-bit nibbles!

If **Keep size** is unchecked and you type more data than you have selected, or size of the data is increased because you have inserted some bytes, OllyDbg will update data beyond the selection. To prevent this, check **Keep size**.

Maximal edit size is limited to 1024 bytes. To move from one control to another, use shortcuts Ctrl+UpArrow and Ctrl+DnArrow.

Assembler

Assembler is similar to the binary edit, and works both on code and data:
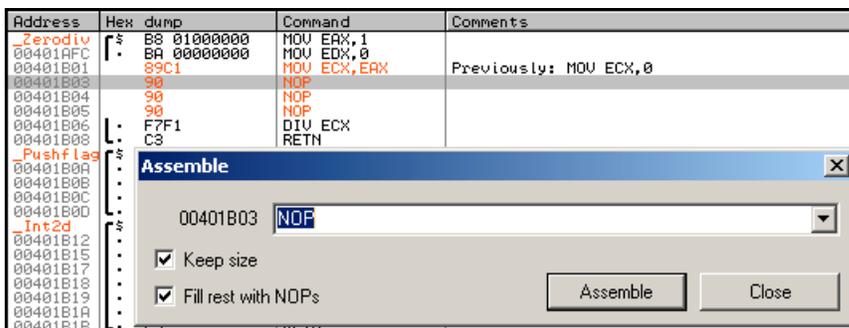
Select block of data you want to modify and call assembler. It will show the first selected command. Edit this command, or type new, and press **Assemble** (or Enter key). If command is invalid, dialog will display an error. Otherwise, command will be written to the memory and selection updated.

If new command does not fit into the integer number of the original commands and **Fill rest with NOPs** is checked, remaining bytes will be filled with code 0x90 (NOP). This is the preferrable way. If option is unchecked, residual bytes of the previous command remain unchanged.

Assembler always attempts to select the shortest binary form. It automatically removes analysis from the modified code.

Binary copy and paste

Binary copy copies selection to the clipboard as a hexadecimal text. For example, operation on the selection:

results in

```
B8 01 00 00 00 BA 00 00 00 00 B9 00 00 00 00 F7
F1 C3
```

Binary paste is strictly limited to selection. If selection is longer, remaining bytes are not modified. Note that paste ignores all characters that are not hexadecimal digits. This is what will happen if you attempt to paste the phrase "HEXADECIMAL CODE":

OllyDbg automatically creates backup of the modified memory blocks. If you want to undo modifications, select it and press Alt+Backspace, or choose **Edit | Undo selection** from the menu.

How to save modifications to executable file

If your modification works, you need to transfer it from the memory back to the executable file. Select modified code or data and choose **Edit | Copy to executable**. OllyDbg creates copy of executable file and applies modification. It even corrects fixups. If fixups are not stripped, try to change code so that fixups remain on the proper locations within the new commands, otherwise modified code may fail if OS loads it to different memory location!

If menu item **Copy to executable** is not available, this means that memory does not map to the executable. Windows usually align sections of the executable file on 4096-byte borders. If code section is shorter than n*4096 bytes, remaining bytes in the memory will be filled with zeros, but they have no counterpart in the executable file. Uninitialized data section are also not part of the executable.

When all modifications are transferred, go to the dump of the executable file and choose **Save file...** from the pop-up menu. You will be asked to select file name. As old file is open by the Debuggee, you must select different name

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 for my 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 this address together with the version of OllyDbg and brief explanation.

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