$#K{bmc bm0.WMF}  OllyDbg v1.10

Read this first

Official stuff

Before you start

Components

How to use OllyDbg

Problems and apologies

$#K# What is OllyDbg?

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

Version 1.10 is a final release. This project is closed and I will no longer support it. But don't be afraid: OllyDbg 2.00, redesigned from scratch, will come soon!

Requirements. Works under Windows 95, 98, ME, NT or 2000, XP (not 100% tested), on any Pentium-class computer, but for a comfortable debugging you will need at least 300-MHz processor. OllyDbg is memory-hungry. If you're going to use extended features like tracing, I recommend 128 or more MB of RAM.

Supported processors. OllyDbg supports all 80x86, Pentium, MMX, 3DNow!, including Athlon extentions, SSE instructions and corresponding data formats. It doesn't support SSE2.

Configurability. More than 100 options (oh dear!) control OllyDbg's behaviour and appearance.

Data formats. Dump windows display data in all common formats: hex, ASCII, UNICODE, 16- and 32-bit signed/unsigned/hex integers, 32/64/80-bit floats, addresses, disassembly (MASM, IDEAL or HLA), or as commented PE header or thread data block.

Help. This file contains essential information necessary to understand and use OllyDbg. If you posess Windows API help (*win32.hlp*, not included due to copyright reasons), you can attach it and get instant help on system calls.

Startup. You can specify executable file in command line, select from menu, drag-and-drop file to OllyDbg, restart last debugged program or attach to already running application. OllyDbg supports just-in-time debugging. Installation is not necessary, you can start OllyDbg from the floppy disk!

Debugging of DLLs. With OllyDbg, you can debug standalone dynamic-link libraries (DLLs). OllyDbg automatically starts small executable that loads library and allows you to call its exports.

Source debugging. OllyDbg reads debugging information in Borland and Microsoft formats. This information includes source code and names of functions, labels, global and static variables. Support for dynamical (stack) variables and structures is very limited.

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 reswitch 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. 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. You may help to improve analysis by supplying hints.

Object scanner. OllyDbg scans object files or libraries (both in OMF and COFF formats),

extracts code segments and locates them in the debugged program.

IMPLIB scanner. Some DLLs export their symbols only by ordinals that are not very meaningful for human eyes. If you have corresponding import library, OllyDbg translates ordinals back into symbolic names.

Full UNICODE support. Almost all operations available for ASCII strings are also available for UNICODE strings, and vice versa.

Names. OllyDbg shows all imported and exported symbols and names from debugging information both in Borland and Microsoft formats. Object Scanner allows to recognize library functions. You can add your own names and comments. If functions in some DLL are exported by ordinals, you can attach import library and restore original names. OllyDbg also knows symbolic names of many 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 2300 frequently used C and Windows API functions and decodes their arguments. You can add your own descriptions, or assign predefined decodings. You may set logging breakpoint on a known 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 actual arguments.

SEH chain. Stack traces and displays the chain of SE handlers. Full chain is available in a separate window.

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 functions that call some procedure or that this procedure calls, for all referenced text strings, for all calls to different modules, for name, for masked binary sequence in the whole allocated memory. If multiple locations are found, you can quickly navigate between them.

Windows. OllyDbg lists all windows created by debugged application and sets breakpoints on window, class or even selected message or group of messages.

Resources. If Windows API function references resource string, OllyDbg will extract and show it. Support for other types is limited to list of attached resources, dump and binary editing.

Breakpoints. OllyDbg supports all common kinds of breakpoints: simple breaks, conditional breaks, breaks that write information (for example, function arguments) to log file, memory breakpoints on write and access, hardware breakpoints (ME/NT/2000 only). In extreme case of hit tracing, INT3 breakpoint can be set on *every* command in the module. On 500-MHz processor under Windows NT, OllyDbg can process up to 5000 breaks per second.

Watches and inspectors. Watch is an expression evaluated each time the program pauses. You can use registers, constants, address expressions, boolean and algebraical operations of any complexity. You can compare ASCII and UNICODE strings. Inspector is a watch that contains up to 2 indexes and can be presented as a two-dimensional table, allowing to decode arrays and structures.

Heap walk. On Win95-based systems, OllyDbg lists all allocated heap blocks.

Handles. On NT-based systems, OllyDbg lists all system handles owned by debugged application.

Execution. You can execute program step-by-step, either entering subroutines or executing them at once. You can run program till next return or to specified location, or animate execution. When application runs, you still have full control over it and can view memory, set breakpoints and even modify code "on-the-fly". At any time, you can pause or restart the debugged program.

Hit tracing. Hit trace shows which commands or procedures were executed so far, allowing you to test all branches of your code. Hit trace sets breakpoint on every specified command and removes it when command is reached (hit).

Run tracing. Run trace executes program step-by-step and protocols execution to a large circular buffer. This protocol contains all registers (except for SSE), flags and thread errors, messages and decoded arguments of known functions. You can save original commands that easies debugging of self-modified code. You can specify the condition to stop tracing – whether an address range, expression or a 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 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. Automatical backup allows to undo changes. You can copy modifications directly to executable file, OllyDbg will even adjust fixups. OllyDbg remembers all patches applied to program in previous debugging sessions. You can apply or remove them with a couple of keystrokes.

Self-extractable files. When debugging SFX file, you usually want to skip self-extractor and stop at entry to original program. OllyDbg implements SFX tracing that attempts to locate real entry. SFX tracing usually fails on protected self-extractors. After entry is found (or specified), OllyDbg can skip extractor more quickly and reliably.

Plugins. You can add features to OllyDbg by writing your own plugins. Plugins access all important data structures, add menus and shortcuts to existing OllyDbg windows and use more than 100 plugin API functions. Plugin API is well-documented. Standard distribution includes two plugins: Command line and Bookmarks.

UDD. OllyDbg saves all program- or 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 the friendly debugger.

$#K What's new in version 1.10

Version 1.10 is a final release. This project is closed and I will no longer support it. But don't be afraid: OllyDbg 2.00, redesigned from scratch, will come soon!

- **Most important new features:**

  - Debugging of standalone DLLs (witout main executable);
  - SEH chain window;
  - Analysis hints that help Analyzer to decode ambiguous code snippets;
  - Logging breakpoints and trace condition support pass counters that tell OllyDbg to pause execution after breakpoint is hit N times or after N commands are added to run trace;
  - On break, conditional logging breakpoints can pass several text commands to plugins, allowing rudimentary automation;
  - You can mark DLLs as system or non-system. Important for Run trace where you may request to trace over calls to system DLLs;
  - New Security option "Save user data outside any module to main .udd file"allows to keep breakpoints and comments that belong to no particular module;
  - Run Trace optionally displays and logs to file modified flags;
  - Maximal length of argument string is changed from 1024 to 4096 characters;
  - You can specify the number of lines visible after current command during stepping and tracing.

- **New plugin functions:**

  - Callback function *ODBG_Paused(int reason,t_reg *registers)* or its extended version *ODBG_Pausedex(int reasonex,int extmode,reg *registers,DEBUG_EVENT *debugevent)*. Called each time when debugged application is paused;
  - Callback function *ODBG_Plugincmd(int reason,t_reg *registers,char *cmd)*. Called when application is paused on conditional breakpoint and this breakpoint contains commands to be passed to plugins;
  - Function *Settracecount(ulong count)* sets number of commands to execute before run trace is paused;
  - Function *Settracepauseoncommands(char *cmdset)* specifies set of commands to pause at;
  - Functions *Getbreakpointtypecount(ulong addr,ulong *passcount)* and *Setbreakpointext(ulong addr,ulong type,char cmd,ulong passcount)* support pass count in conditional breakpoints;
  - Function *Listmemory()* actualizes list of memory blocks.

- **Bugfixes:**

  - Registers menu option "Copy all registers to clipboard" now copies EAX, too;
  - In code with non-standard alignment of sections, analysis and displayed code were desynchronized, so click on some line selected different line;
  - Menu "Follow in Dump" now displays more intuitive items if one of operands is implicit stack location (like in PUSH);
  - OllyDbg now correctly disassembles VxDCall and VxDJump used by Win95 drivers. However, it assembles them to the same code. In almost improbable case that anybody will use this pseudocommand, user must correctly set bit 0x00008000 to distinguish between call and jump;
  - OllyDbg accepted nonsense like REP STOS FWORD [EDI] and compiled it to REP STOS DWORD [EDI];
  - Names window sometimes lost its contents after new modules were loaded;
  - Assembler supports simplified form of IMUL: IMUL reg,const (disassembled as IMUL

reg,reg,const);
- Disassembler used address size instead of operand size to decode size of immediate offset (JMP FAR ssss:oooooooo);
- Tabs in source text in Disassembler comments and info pane were displayed as small rectangles. Now they are extended to 8 spaces;
- ARPL was decoded with 32-bit size of operands (correct decoding is ARPL r/m16,r16);
- OllyDbg now should correctly work in multi-monitor configurations (but I am unable to verify this);
- 2-byte INT 3 (CD 03) was processed incorrectly;
- Column "Handle" is removed from Threads window. This column falsely displayed handle assigned to thread in OllyDbg. Such handles are meaningless in the contents of debugged application;
- Analysis crashed on large modules due to overflow of jump-tracing table;
- Non-standard modules (with size not aligned on 4096 bytes) lost all user-supplied information, like breakpoints or comments;
- On attempt to step over call to ExitThread() or ExitProcess(), OllyDbg attempted to set INT3 breakpoint on next command which in some cases was data;
- Request to flush gathered run trace data to file wrote invalid commands;
- OllyDbg hasn't checked that *.udd* directory specified in .ini file really exists;
- After binary edit, Disassembler haven't updated selection, so that it was possible that some command is only partially selected;
- OllyDbg crashed when it received command line in form "a.exe %.622496x" or similar that looked like format specifiers;
- Dump windows underlined fuxups outside the dumped memory area;
- Hex edit window behaved unpredictably when user moved from one presentation to another and some characers were incomplete;
- FSAVE/FRSTOR and FLDENV/FSTENV displayed invalid operand size when used with prefix 66;
- Analyser hanged (forever or for several minutes) if function with insufficient number of arguments was placed close to the beginning of the memory block;
- Cosmetical: "Save file" window called *GetOpenFileName()* instead of *GetSaveFileName()*;
- ESP was not logged to file opened in Run trace, even if corresponding option was set;
- Given (invalid) command MOV QWORD [1234],0 (or many others between memory location and constant), Assembler bravely compiled it to code with 8-byte immediate constant;
- When opening executable with quoted argument line (a.exe "ab" "cd") for the second time, OllyDbg removed outer quotes (a.exe ab" "cd). This was not my fault, honestly, but a misfeature of *GetPrivateProfileString()*!

What was new in version 1.09c

- **New features:**
  - When stepping or animating, Disassembler window attempts to leave 1 or 2 completely visible strings below current command;
  - Run trace saves 16 high-order bits of flag register;
  - New global shortcut Ctrl+P opens Patches window;
  - OllyDbg exports two new functions: *int Attachtoactiveprocess(int newprocessid)* and *HWND Createpatchwindow(void)*;

- **Bugfixes:**
  - PEXTRW swapped its two operands (MMX and general-purpose register);
  - Some error messages were covered by main window when this was set always-on-top;
  - OllyDbg recognized some absolutely correct PE files as bad due to unhappy section placement;
  - LOCK not allowed with reg-reg and if no write to memory;

- Assembler reported invalid mnemonics on IN command, because scanner mixed mnemonics with operator IN;
- Added support for non-standard short PE Optional Header;
- When paused on hardware breakpoint, OllyDbg was unable to step over some commands if automatical hardware breakpoints were allowed;
- Too long program arguments (longer than 256 bytes) caused OllyDbg to crash due to buffer overflow;
- Command LEA with 16-bit addressing reported 'Superfluous prefix' because it was marked as not accessing memory;
- NEAR/FAR modifiers were highlighted with random colours;
- Problems converting Japanese UNICODE text to multibyte (not checked because I don't know Japanese);
- Commands SETZ, SETO... with unused Reg field of ModRegRM byte not equal to 0 were not recognized. Now OllyDbg warns if option "Non-standard command forms" is not active;
- OllyDbg assembled and disassembled invalid command MOV CS,R16 without warnings;
- If you pressed Alt+F2 (or X on toolbar) but then decided not to close debugged process, OllyDbg nevertheless removed all process data, making debugging impossible;
- If size of executable code was shorter than size of section or size of module shorter than 1 memory block, analysis disappeared when scrolling code.

What was new in version 1.09b

- **New features:**
  - Patch manager is perharps the most important new feature. OllyDbg remembers all patches applied to debugged application in previous debugging sessions. From the Patch window, you can quickly apply patches or restore original code;
  - With one command, you can copy all patches in a module to executable file;
  - One MDI window may be declared as "always on top". Attention, in order to support this feature, plugins must pass WM_WINDOWPOSCHANGED to *Tablefunction()*;
  - You can specify directories where OllyDbg saves .udd files and searches for plugins;
  If selected command is a jump destination, OllyDbg can display "jump from" path;
  - On NT-based systems, Handles window displays list of handles owned by debugged application;
  - If command that you type in Assembler dialog contains comment, it will be automatically added to the command;
  - You can reswitch between debugging options and appearance without closing options dialog;

- **Improved analysis:**
  - "Search for all intermodular calls" includes now predicted calls;
  - Option to trace registers in the whole procedure. Previous analyzer predicted registers only within linear pieces of code (without jumps from outside);
  - Option that tells Analyzer that unknown functions preserve registers EBX, ESI and EDI. If this is not true, contents of registers may be predicted incorrectly, so use this option with care.

- **Bugfixes:**
  - In call tree, OllyDbg temporarily forgot calls predicted in previous debugging session with register tracing;
  - Sometimes OllyDbg created new .udd file (xxx_1, xxx_2 etc.) after each debugging session.

What was new in version 1.09

- **New features:**
  - Additionally to MASM and IDEAL disassembling modes, version 1.09 supports also HLA syntax (High Level Assembly, developed by Randall Hyde). HLA is public domain software, you can download it together with documentation and sources from http://webster.cs.ucr.edu;
  - Analyzer knows that there is no return from calls to *kernel32.ExitThread()* and *kernel32.ExitProcess()* and interpretes them as end of procedure;
  - If several executable modules have same short 8-byte name, OllyDbg renames them to xxx_1, xxx_2 etc;
  - To avoid mixing of .udd files in cases when main file and DLL have same name, or if program uses DLLs with same name that reside in different directories, OllyDbg adds _1, _2 etc. to names of .udd files. This feature is active if option "Security|Ignore path and extention" is unchecked;
  - Option to synchronize CPU with source;
  - OllyDbg supports relative pathes to source files in Borland's debugging information generated by BCC5.5;
  - Debugging engine now can step into unknown commands, like SSE2 (new Security option);
  - Option to lock stack (i.e. stack window doesn't scroll when stepping);
  - Register window displays debug registers DR0..3,6,7. Debug registers are not saved to run trace and you can't modify them. Caveat plugin writers: size of structure t_reg is changed!
  - From the dump of executable file, you can jump to it's memory image in Disassembler or CPU Dump;
  - OllyDbg recognizes "real" (undocumented) SAL instruction but, in accordance to Intel's documentation, assembles it to SHL. Both instructions have same effect;
  - New undocumented opcode: ICEBP (INT1);
  - Search for address and binary string in stack - now officially;
  - Option to save width of columns to .ini file;
  - Additionally to jumps, CPU info pane, list of known jumps and corresponding menus display local (intramodular) calls to selected instruction;
  - If you browse cases, jumps or calls to location in dialog, Disassembler jumps to corresponding commands as you change selection. On Cancel, old selection is restored.

- **Bugfixes:**
  - If you close debugged program (Alt+F2), OllyDbg now correctly closes all associated handles. Open handles made recompilation of executable file impossible;
  - When file name contained spaces, under some circumstances symbols after space were interpreted as parameters in command line. This explains, for example, the great mystery of disappearing patches;
  - OllyDbg correctly attaches to active process from Task Manager. Caveat: format of JIT record in registry is changed (added quotes arond file name), so new version will not recognize old JIT declaration;
  - Short (no-operand) forms of INS and OUTS now recognized as I/O commands;
  - Corrected invalid decoding and assembling of SSE instructions MOVHLPS and MOVLPS. Intel made it hard: MOVHLPS is a register-register and MOVLPS is a register-memory form of the same command, and they behave differently...
  - Sometimes OllyDbg crashed on "Execute till return". Heer I forgot to check for a possible NULL pointer;
  - Even when main module resides in system directory, it is considered now user code;
  - And additionally, several cosmetical improvements.

What was new in version 1.08

OllyDbg 1.08 is the next "stable" version after 1.06. I will treat odd versions as experimental and post them only on my Internet site. (You are free, however, to download and redistribute them).

Registered OllyDbg users will get notified on every even version.

Full list of changes I've made after 1.06 is too long to place it here. The most significant changes are:

- **New features:**
  - Support for debugging information in Microsoft formats, as implemented by *dbghelp.dll*. This includes CodeView, COFF, PDB and SYM formats and (hopefully) any other formats that will be implemented in the future;
  - Call stack backtraces nested calls even if procedures use non-standard prologs and epilogs;
  - Call tree that for every analyzed procedure shows which functions it calls and from which points it's called. Additionally, Call tree recognizes recursive (that call itself, directly or indirectly) and pure functions (that make no calls and doesn't modify memory operands except for local variables on stack);
  - Heap list recognizes all memory blocks allocated on the heap. Unfortunately, this option is not available on NT-based systems (NT, 2000, XP?);
  - Code highlighting. You can highlight different types of commands, such as FPU/MMX/SSE, jumps and conditional jumps, pushes and pops, calls, returns, privileged, bad and filling commands. Optional highlighting of operands recognizes general, FPU/SSE and segment/system registers, memory operands on the stack (i.e. accessed via ESP or EBP) and in ordinary memory. Additionally, it differentiates between constants that are valid memory addresses and all other constants. You can create custom highlighting schemes;
  - List of windows displays basic window information (class and window function, parent, styles) and allows to set breakpoints on class, single window or on selected messages or message groups.
  - You can search for all intermodular calls, including indirect (for example, loaded with GetProcAddress()), and set breakpoints on all calls to selected function at once. Command line plugin uses this feature to implement long-awaited "bpx API" command.

- **Strongly improved analysis:**
  - Recognition of procedures is significantly accelerated;
  - Recognition of loops. By definition, loop is a closed sequence of commands (where last command is a jump to the first) with a single entry point and any number of exits. Notice that entry is not necessarily the first command in loop;
  - Recognition of switches. Moreover, OllyDbg even attemts to suggest the meaning of separate cases;
  - Analyzer recognizes RETs misused as JMPs (but only if there are no intermediate PUSH, POP or CALL), pushes of 4- and 8-byte floating point numbers like ADD ESP,-4; FSTP [DWORD SS:EBP] and procedures that serve as structured exception handlers;
  - Jump information is kept in .udd file between sessions, supporting instant list of jumps to the given address ("Jumps from");
  - Analyzer automatically extends function with variable number of arguments (ellipsys).

- **Security:**
  - OllyDbg attempts to restore size of code section when both code size and offset in PE file header are 0.
  - Access to memory containing active INT3 breakpoint (reading or writing) and command CLI considered suspicious during run trace;
  - OllyDbg warns if user has no administrative rights under NT;
  - If *psapi.dll* or *dbghelp.dll* distributed with OllyDbg is older than installed in system directory, OllyDbg recommends to delete local version.

- **Rare or undocumented commands and aliases:**
  - AMD-specific instructions SYSCALL and SYSRET;
  - Undocumented instruction FFREEP STn;

- Assembler understands alias mnemonics JNAE, JAE and SAL.

- **Expressions:**
  - Expressions allow for several thousands symbolic constants, like WM_PAINT or O_RDONLY;
  - New arithmetical operation 'IN' in expressions allows for easy specification of ranges;
  - New pseudofunction WINTEXT (alias: GetWindowText) in expressions returns ASCII text of window with specified handle. Notice that this text is treated as a string constant, so comparison is to full length but ignores case;
  - Watch window highlights syntax errors and selects location of error in expression.

- **Other improvements:**
  - Stack pane recognizes chain of structured exception handlers;
  - When address has both symbolic name and ordinal, symbolic is preferred;
  - Jump path is grayed if conditional jump is not taken;
  - Automatic wakeup of thread that executes WaitMessage();
  - Profile for the whole module or for the whole application, option to group adjacent commands in profile or count each command separately;
  - Dump window supports direct copying of modifications to executable file;
  - Modules window shows version of executable file;
  - Binary search dialogs support 10-item history of previous search strings;
  - Multi-instance windows (like standalone Dump or Names) remember last selected appearance (font, colours and scrollbar). Memory and file dumps have different appearances;
  - Execution till user code (shortcut: Alt+F9) allows to return back to debugged application from the system code. DLL is considered system if it resides in system directory;
  - Options to pause program on system breakpoint, module entry point or WinMain();
  - Option to kill the thread;
  - Option to analyze all modules at once;
  - Support for code/data selection with Shift+LeftClick.

- **Many corrected bugs, among them:**
  - SFENCE/LFENCE/MFENCE commands were decoded incorrectly;
  - FXSAVE/FXRSTOR commands caused general protection fault when OllyDbg attempted to decode 512-byte memory operand;
  - Assembler reported error on attempt to assemble SETcc;
  - FSTSW AX compiles improperly (with 16-bit prefix);
  - In Reference window, sometimes it was not possible to follow address in Disassembler;
  - Font (all)/Colours (all) didn't work from CPU Dump;
  - OllyDbg sometimes locked on attempt to terminate debugged process that was stopped on exception;
  - After Ctrl+G in Debug window, contents of window was not updated (but corresponding menu item worked correctly);
  - Text constants in expression incorrectly interpreted symbols '\\', '\n' etc.
  - Several other, not so important bugs or misfeatures.

$#KTips and tricks

- You can use OllyDbg as a binary editor. Choose View, File and select the file you want to dump. Size of files, however, is limited by amount of free memory.
- If you have modified the dump of executable file and want to undo some changes but forgot to create backup, load original file as backup copy. Now you can search for modified data.
- Scan object files before you start analysis. In this case OllyDbg will decode arguments of known C functions.
- Some tables contain hidden data. Try to increase width of columns.
- All dump windows (including Disassembler) can display relative addresses, simply doubleclick the address.
- You can scroll all dump windows one byte at a time. Use Ctrl+UpArrow or Ctrl+DnArrow.

$#K Apologies

"We apologize for inconveniences"
D. A.

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.

$#KProblems with localized versions of operating system

I expect no problems if you run OllyDbg on the non-English (American) version of the Microsoft Windows operating system, but haven't performed any tests. OllyDbg uses English (I hope understandable) language. In some cases, however, it asks help from the OS, like in decoding of resource language identifiers etc. As a result, sometimes you can see a weird mix of english and native words. If native words are indecipherable, try using UNICODE font (Courier or Lucida).

Some users reported that OllyDbg doesn't work on the localized versions of Windows XP. I was unable to reproduce this behaviour.

$#KMinimal system requirements

To start and run OllyDbg, you need at least:

- 586 processor (166 MHz or higher recommended)

- Microsoft Windows 95 (OSR2), Windows 98, Windows ME, Windows NT 4.0 (Service Pack 2+), Windows 2000 or Windows XP. (Names are used for identification purposes only, assume as many ® symbols as you can apply)

- 64 MB physical memory (128+ MB recommended)

- at least 1 MB free disk space

- Screen resolution 800x600 (1024x768 or higher is strongly recommended)

- Mouse or compatible pointing device (required)

To debug programms on Windows NT, 2000 or XP, you may have to have administrative rights.

OllyDbg is unable to debug .NET applications.

Win32s (32-bit extension for Microsoft Windows 3.1) doesn't support all important debugging functions. OllyDbg will not work under Win32s.

See also: Problems with localized versions of operating system

$#KInstallation

OllyDbg requires no installation. Simply create new folder and unpack archive Odbg110.zip to this folder. If necessary, drag-and-drop ollydbg.exe to the desktop to create shortcut.

To run OllyDbg on Windows NT 4.0, you need *psapi.dll*. To read symbolic debugging information in Microsoft formats (CodeView, COFF, PDB and SYM), you need *dbghelp.dll*. These files are Microsoft Distributable files and are included in the archive. Some Windows installations already contain these libraries. If OllyDbg detects that existing library has higher version or that *psapi.dll* is not required, it asks you to delete unnecessary files from the OllyDbg's directory.

Some 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.

See also: Your privacy and security

$#K Support

The available support is limited to the Internet page http://home.t-online.de/home/Ollydbg. From here you will be able also to download bugfixes and new versions. To receive notifications on available upgrades, please register. 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**.

I've made the source codes of Disassembler (version without analysis and SSI support) and Assembler (without SSI support) available for free under GPL license. You can download them from my page.

Full source code is available but will cost you some money. See, I've spent more than a year to write these 3.2+ MB of dense well-documented code and data and want something in return. 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.

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

$#K Registration

OllyDbg 1.10 is Copyright (C) 2000-2004 Oleh Yuschuk. This software is not a freeware. To use this program on a permanent basis or for commercial purposes, you should register it. The registration is **free of charge** and assumes **no financial or other obligations** from your side - just be fair and let me know that you like this software. Any personal data in the registration form is optional (use your nickname or pseudonym if you want).

If you use OllyDbg together with **Randall Hyde's HLA (High Level Assembly)**, you don't need (but still allowed) to register.

When registering, you can subscribe for information (email) on the new release versions of this program. In this case you agree not to treat this information as a spam as long as number of letters does not exceed 4 each calendar year and they contain no advertisements from the third parties. If you no longer want to receive this information - well, just let me know, and I will immediately delete your address from my database.

If you are already a registered OllyDbg user, you don't need to re-register this version. If you are new, please read license argeement, fill the registartion form (*register.txt*) or copy and fill the following section from the help and email it to Ollydbg@t-online.de. I will keep your information confidential and will not give it to third persons, unless forced by a law.

```
                Registration form for OllyDbg v1.10


To use OllyDbg, you must agree with  all  of  the  terms  and
conditions of the accompanying License  Agreement. All  other
answers are optional.

Name      _____________________________________________________

Title     _____________________________________________________

Company   _____________________________________________________

City,state_____________________________________________________

Country   _____________________________________________________

Where did you find OllyDbg ____________________________________

          _____________________________________________________

Are you going to write your own plugins

          (____) Yes    (____) No    (____) Don't know

I agree with all the terms and condition of the  accompanying

License Agreement (Very important! Please mark!)

          (____) Yes    (____) No

Date of registration __________________________________________
```

If you want to receive notifications when OllyDbg 2.00 and subsequent versions will be ready, please enter your email address here:

_____

Thank you. If you have ideas how to improve OllyDbg and make it easier in use, or want to have some new features, please let me know. Your opinion helps me a lot!

Your first idea: _____

_____

Your second idea: _____

_____

Your third idea: _____

_____

$#KLegal 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 1.10, OllyDbg Plugin Development Kit version 1.10 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) 2000-2004 Oleh Yuschuk. To use this Software on a permanent basis or for commercial purposes, you must register it by filling the supplied registration form and sending it to the Author. You don't need to register Software if you use it exclusively with Randall Hyde's High Level Assembly. If you are already a registered OllyDbg user, you don't need to re-register the Software again. If the Software is registered to a company or organization, any person within the company or organization has the right to use it at work. You may install the registered Software on any number of storage devices, like hard disks, floppy disks 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 (for example, plugins) 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 or diskette), even if your distribution contains additional files.

You are allowed to develop and distribute your own plugins -- Dynamic Link Libraries that connect to the Software and make use of the functions implemented in the Software -- free of charge provided that a) your plugins contain no features that persuade or force user to register them, or limit functionality of unregistered plugins; b) you allow free distribution of your plugins on the conditions similar to that of the Software; and c) you collect no fee (except for transport media, like CD or diskette). If you want to develop commercial plugin, please contact Author for a special Agreement.

The distribution includes files PSAPI.DLL and DBGHELP.DLL that are the Microsoft(R) Redistributable files. These files should be installed only in the directory where the Software resides. You should use supplied PSAPI.DLL only on Windows NT(R) 4.0. You are not allowed to distribute PSAPI.DLL and/or DBGHELP.DLL separately from the Software.

This Agreement covers only the actual version 1.10 of the OllyDbg and version 1.10 of the OllyDbg Plugin Development Kit. All other versions are covered by 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!

$#K Your privacy and security

The following statements apply to versions 1.00 - 1.10 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 6 keys:

HKEY_CLASSES_ROOT\exefile\shell\Open with OllyDbg
HKEY_CLASSES_ROOT\exefile\shell\Open with OllyDbg\command
HKEY_CLASSES_ROOT\dllfile\shell\Open with OllyDbg
HKEY_CLASSES_ROOT\dllfile\shell\Open with OllyDbg\command
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;

- 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;

- (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.

If you need similar privacy and security statement concerning OllyDbg plugins, please contact authors of the corresponding plugin. I assume no responsibility for any third-party plugins. As the source code of bookmark.dll is freely available, I also assume no responsibility for this plugin unless downloaded directly from my Internet site.

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.

$#KGeneral principles

I expect that you are familiar with the internal organization of 80x86-compatible processors and have experience of writing programs in Assembler. I also expect that you are familiar with Microsoft Windows.

OllyDbg is a single-process, multi-thread analysing code-level debugger for 32-bit programs and dynamic-link libraries (DLLs) running under Windows 95, Windows 98, Windows ME, Windows NT and Windows 2000. It allows you to debug and patch executable files in PE (Portable Executable) format. "Code-level" here means that most of time you work with low-level bits, bytes and processor commands. OllyDbg uses only documented Win32 API calls, so the chances are good that you will be able to use it on the next derivatives of 32-bit Windows operating systems. OllyDbg works under Windows XP, but I've made no exhaustive tests and therefore can't guarantee the full functionality. Note that OllyDbg can't debug .NET applications.

OllyDbg is not compiler-oriented. It contains no special rules telling which code sequences this or another compiler will generate under some circumstances. As a result, you can use OllyDbg equally good with any code generated by any compiler translating any non-interpreting language, or with code written directly in Assembler.

OllyDbg works in parallel with the debugged application. You can browse code and data, set breakpoints, stop or resume threads and even modify memory without pausing the execution. (Sometimes this is referred to as a soft debugging mode, in contrast to the hard mode where debugger is blocked when application is running and vice versa). Of course, if requested operation is not atomic, OllyDbg will stop application for a short time, but this pause is transparent for you.

Sometimes unexpected crash happens when application is running without debugger. You can register OllyDbg as a just-in-time debugger, and it will attach to the program and point to the place where exception occured.

With OllyDbg, you can debug standalone DLLs. OS can't start DLL directly, so OllyDbg keeps, as a packed resource, small application that loads your library and allows you to call exported functions with up to 10 arguments.

OllyDbg is strongly module-oriented. Module is the main executable file (usually with extension .EXE) or dynamic-link library (usually .DLL) loaded either on startup or dynamically on request. During debugging session you set breakpoints, define new labels and comment assembler commands. When some module unloads from the memory, Debugger saves this information to file with the same name as the debugged module and extension .UDD (stays for User-Defined Data). Next time when OllyDbg loads this module, it automatically restores all debugging information, no matter which program uses this module. Suppose you are debugging *Myprog1* that uses *Mydll* and set some breakpoints in *Mydll*. Then you start debugging *Myprog2* that also makes use of *Mydll* - and all breakpoints in *Mydll* are still here, even if *Mydll* loads on different location!

Some debuggers treat memory of debugged process as a single (and mostly empty) arena 2**32 bytes in size. OllyDbg takes another approach. Here, memory consists of several 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. Such cases, however, are seldom.

OllyDbg is a memory-hungry application. It allocates up to 3 MB of memory directly on startup and another megabyte or two when you first load debugged program. Each analysis, backup, trace or file dump take their own pieces, so please don't panic when you are debugging larger

project and Program Manages shows 40 or 60 allocated megabytes.

To effectively debug some program without source code, you must first understand how it works. OllyDbg contains plenty of features that facilitate this understanding.

First of all, it contains built-in code analyzer. Analyzer walks through the code, separates commands from data, recognizes different data types, procedures, decodes arguments of standard API functions (about 1900 most frequently used) and attempts to guess number of arguments of unknown functions. You can add your own function descriptions, too. It marks entry points and jump destinations, recognizes table-driven switches and pointers to strings, adds some comments and even shows the direction of jumps. Basing on the results of analysis, call tree shows which functions given procedure calls (directly or indirectly) and recognizes recursive, system and leaf procedures. If necessary, you may set decoding hints that help analyzer to decode ambiguous pieces of code or data.

OllyDbg also includes Object Scanner. If you possess libraries or object files, Scanner will locate library functions in the debugged application. Calls to standard functions take significant part of all function calls (up to 70%, according to my estimations). If you know which function is being called, it will not distract your attention and you can simply step over the call. Analyzer knows by name more than 400 standard C functions, like fopen or memcpy. I must admit however that OllyDbg in actual version is unable to locate short (consisting of hardly more than return) or similar functions (which differ only in relocations).

Object scanner also understands import libraries. If some DLL exports functions by ordinals, you will see meaningless cryptical numbers instead of function names. Developers of such DLLs usually supply import libraries that set correspondence between symbols and ordinals. Tell OllyDbg to use import library, and it will restore original names.

Object-oriented languages, like C++, use technique called name mangling that adds information on types and arguments to symbol. OllyDbg can demangle such names, making program more readable.

OllyDbg fully supports UNICODE. Almost everything you can do with ASCII string is possible with UNICODE, too.

Assembler commands are all very similar. It often happens that you don't know whether you've already traced this branch of code or not. In OllyDbg you can add own labels and write comments. This greatly facilitates debugging. Notice that once you commented some DLL, comments and labels will be available each time DLL is loaded - even if you debug different application.

OllyDbg traces standard stack frames (those created by PUSH EBP; MOV EBP,ESP). Modern compilers have option that disables generation of stack frames, in this case stack walk is not possible. When execution reaches call of known funbction, stack window decodes its arguments. Call stack window displays sequence of calls leading to the actual point of execution.

Modern OO applications extensively use structured exception handling (SEH). SEH window displays chain of exception handlers.

Many different search options allow you to find binary code or data, command or a sequence of commands, constant or string, symbolic name or a record in a run trace.

For any address or constant, OllyDbg can prepare the complete list of commands referencing this address or constant. You then walk this list and decide whether given reference is of any importance for you. For example, some function may be called explicitly, or optimizing compiler can load its address in register and then call function indirectly, or push address into the stack as a parameter - not a problem, OllyDbg will find all such places. It will even find and list all relative

jumps and switches to the specified location (*Really? Oh dear!..*)

OllyDbg supports all standard kinds of breakpoints - unconditional and conditional, memory (on write or access), hardware or on access to the whole memory block (the last two work only under Windows ME, NT, 2000 or XP). Conditional expressions can be very complex („break when bit 2 in [ESP+8] is set and word at location 123456 is less than 10 or EAX points to UNICODE string that begins with 'ABC', but skip first 10 breaks"). You may specify one or several commands that OllyDbg passes to plugins when application pauses. Instead of pausing, you can request logging the value of another expression (with brief explanation) or logging the arguments of function known to OllyDbg. On Athlon 2600+ running under Windows 2000, OllyDbg can process up to 25000 conditional breaks per second.

Another useful feature is tracing. OllyDbg supports two kinds of tracing: hit and run. In the first case, it sets breakpoint on every command in specified range (for example, in the whole executable code). After command is reached, it removes breakpoint and marks command as hit. This allows to check at a glance whether some branch of code was executed or not. Hit tracing is astoundingly fast, after short startup application reaches almost full speed. As INT3 breakpoint may have disastrous effect when set on data, I recommend that you don't use fuzzy recognition of procedures. Hit tracing is not available at all when code is not analyzed.

Run trace executes program step by step and records exact execution history together with contents of all registers, known arguments and optionally commands (this helps when code is self-modified). Of course, this requires plenty of memory (depending on the mode, from 15 to 50 bytes per command) but allows for precise backtracing and analysis. You can run trace only selected pieces of code, or even single commands, or you can skip quasi-linear pieces that are of no special interest. For each address, OllyDbg calculates number of times this address appears in run trace log, implementing slow but comprehensive profiling. For example, special command lets you set run trace breakpoint on every entry to recognized procedure, then profile gives you notion how frequently each routine is called. Run trace may automatically pause on command, address range or counter.

OllyDbg automatically manages threads in multithread applications. If you step or trace your program, it automatically resumes current thread and suspends all others. If you run it, OllyDbg restores previous thread state.

You can make momentary snapshot (called backup) of the memory block. OllyDbg then highlights all changes. You can save backup to file and read back, thus locating the differences between two runs. You can view backup, search for next modified piece, or undo all or selected changes. Patch manager remembers all patches applied to debugged application in previous sessions and can apply them again.

You can easily transfer your patches back to the executable file. OllyDbg will automatically correct fixups.

You can’t use OllyDbg on 16-bit Windows with Win32s. This 32-bit extender does not implement all necessary debugging functions.

You can debug neither DOS nor 16-bit NE (New Executable) files, and I have no plans to support this in the future. R.I.P., old good command prompt!

$#K How to start debugging session

The simplest way is to start OllyDbg, choose **File|Open** and select the program you want to debug. If this program requires any command-line arguments, type in the arguments in the box on the bottom of the dialog or select one of the argument lines you've used in the previous sessions.

OllyDbg can debug stand-alone DLLs. In this case it creates and starts small application that loads library and, on your request, calls exported functions.

If you want to restart the last debugged program, simply press **Ctrl+F2** (this is a shortcut for Restart Program), and OllyDbg will run it with the same arguments. Alternatively, choose File and select program from the history. You may also drag executable file or DLL from the Windows Explorer and drop it into OllyDbg.

Of course, you can specify name of the debugged program and its arguments in the **command line** when launching OllyDbg. For example, you can create desktop shortcut pointing to OllyDbg, select Properties, go to Shortcut and add the name of the program to the target line. Each time you double-click this shortcut, OllyDbg will automatically start your program. Note that DLLs don't support command line.

You can **attach** OllyDbg to the running process. Choose File|Attach and select the process from the list. Notice, however, that after you close OllyDbg, this process will terminate. Never try to attach to the system process, this may result in the complete crash of operating system. (To be honest, in most cases OS doesn't allow you to attach to the sensitive process).

#OllyDbg can act as a **just-in-time** debugger. This requires registration in the system registry. Choose Options|Just-in-time debugging and in appearing dialog press "Make OllyDbg just-in-time debugger". Now, when some application crashes, you will be asked whether you want to debug it. Operating system will then start OllyDbg that will stop directly at the point where exception occured. Or, if you've selected attaching without confirmation, OllyDbg will start without bothering to ask. To restore previous just-in-time debugger, press corresponding button in the mentioned dialog, that's all.

#Yet another possibility is to add OllyDbg to the pop-up menu associated with executable files in **Windows Explorer**. (This idea belongs to Jochen Gerster). In the main menu, select Options|Add to Explorer, then press "Add OllyDbg to menu in Windows Explorer". Afterwards you can right-click executable file or DLL in any file list and select OllyDbg from menu. This option creates 4 registry keys:

HKEY_CLASSES_ROOT\exefile\shell\Open with OllyDbg
HKEY_CLASSES_ROOT\exefile\shell\Open with OllyDbg\command
HKEY_CLASSES_ROOT\dllfile\shell\Open with OllyDbg
HKEY_CLASSES_ROOT\dllfile\shell\Open with OllyDbg\command

OllyDbg can debug console (text-based) applications.

OllyDbg can't debug .NET applications. .NET programs consist of pseudocode that Windows interpretes or compiles on-the-fly to native '86 commands.

Notice that if you run Windows NT, 2000 or XP, you may have to have administrator rights to debug programs.

$#K# Debugging of stand-alone DLLs

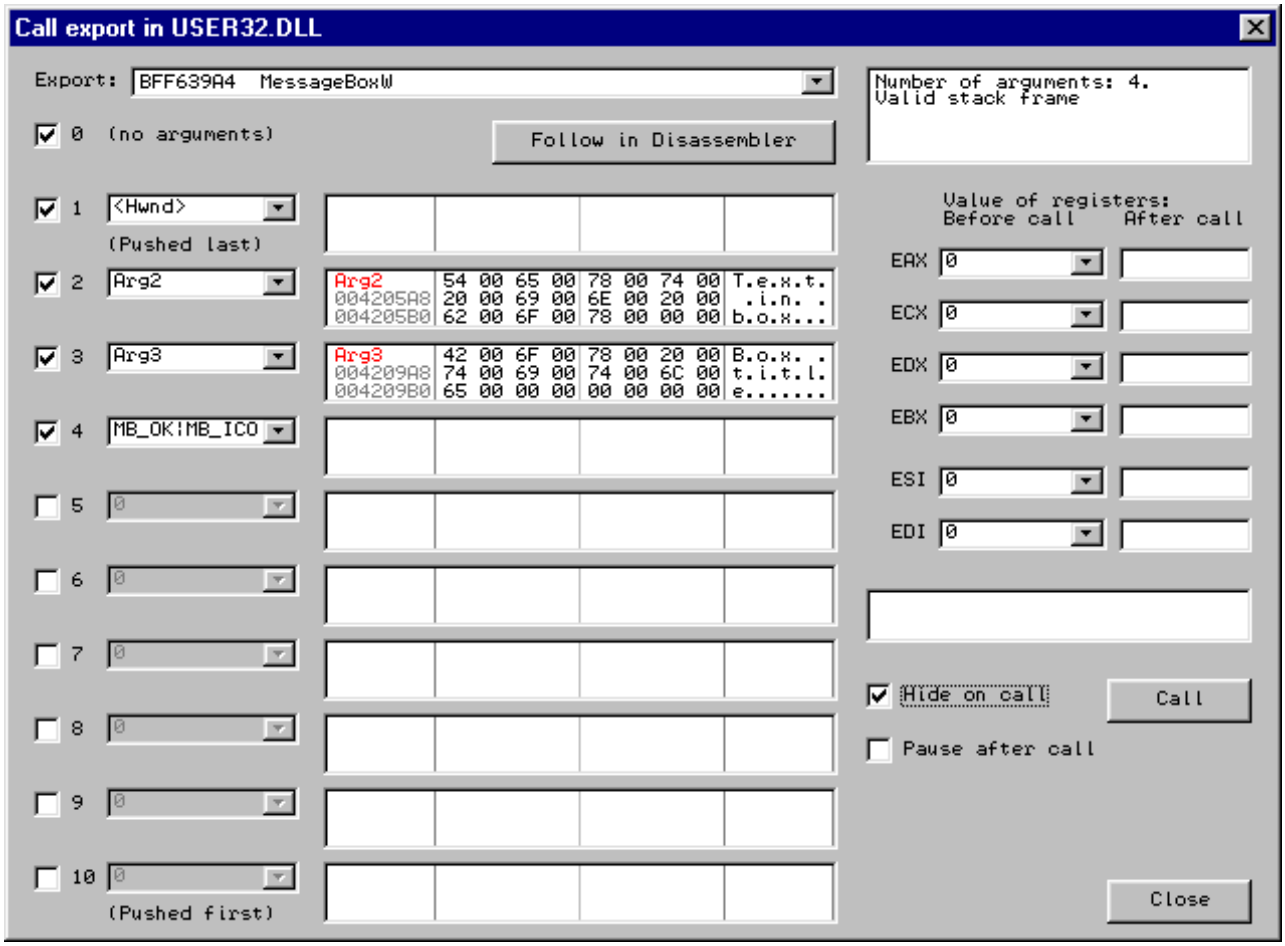
Dynamic-link libraries contain functions callable from other modules and can't be executed directly. To debug DLL, OllyDbg extracts and starts small program that loads DLL and allows you to call exported functions with up to 10 arguments.

This program, *loaddll.exe* (source code available), is kept as a packed resource. OllyDbg extracts it only if file named *loaddll.exe* is absent in OllyDbg's directory. *Loaddll* contains large patch areas where you can place additional task-specific code. If you no longer need this code, delete *loaddll.exe* and next time OllyDbg will extract clean copy.

Open DLL, or drop it into OllyDbg from Explorer. OllyDbg asks you for confirmation and passes full name of DLL as a parameter to *loaddll.exe*. After library is loaded, it pauses at the startup code (<*DllEntryPoint*>). You may set breakpoints, run or trace startup code and so on. After initialization is finished, application pauses again, this time in *loaddll* at a breakpoint labelled **Firstbp** that immediately precedes main Windows loop.

Now you can call DLL functions. From the main menu, select "Debug|Call DLL export". The appearing dialog is non-modal, so you still have full access to all OllyDbg features. You can browse code and data, set breakpoints, modify memory and so on. Select the function you want to call. As an example, here I will use *MessageBox* from *USER32.DLL*. Caveat: *loaddll.exe* already uses this library, so Windows assumes that DLL is already initialized and doesn't call entry point again. Name *MessageBox* is generic, in reality there are ASCII version *MessageBoxA* and UNICODE version *MessageBoxW*. Let's try the second one:

{bmc
bm1.WMF}



As we select function's name, rectangle to the right says: "Number of arguments: 4". Analyzer determined that function ends with RET 10 and correctly recognized number of parameters. RET nnn is typical for functions that use PASCAL calling convention (parameters are passed on the stack, first parameter is pushed last, function removes parameters after call). Most Windows' API functions are PASCAL-style.

Next, set number of stack arguments. In our case this is not necessary, because OllyDbg already knows correct number of arguments. But, of course, you can override this decision anytime by clicking on the corresponding checkbox to the left.
Now fill list of parameters. This dialog supports up to 6 register parameters and up to 10 parameters in stack. Parameter is any valid expression that doesn't use contents of registers. If operand points to memory, Dump window to the right from the argument displays contents of this memory. *Loaddll.exe* reserves 10 memory buffers, 1 K each, labelled as **Arg1** .. **Arg10**, that you can freely use for any purpose. Additionally, dialog supports two pseudovariables: handle of parent window **<Hwnd>** created by *loaddll.exe* and handle of *loaddll*'s instance **<Hinst>**. For your convenience, when you use Call export for the first time, OllyDbg adds them to history lists.

MessageBoxW expects 4 parameters:

- handle of owner window. Here, we simply select **<Hwnd>**;
- address of UNICODE text in message box. Select **Arg2** and press Enter. Dump displays contents of memory buffer in hexadecimal format. This buffer is initially filled with zeros. Select first byte and press Ctrl+E (or, alternatively, choose "Binary|Edit" from menu). In the appearing window, type in UNICODE format "Text in box" or any other text to display;

- address of UNICODE title of message box. Select **Arg3** and write "Box title" to pointed memory;
- style of message box as a combination of MB_xxx constants. OllyDbg knows them, type here MB_OK|MB_ICONEXCLAMATION.

We need no register arguments.

Now we are ready to call export. Option "Hide on call" means that dialog box should disappear from the screen when function executes. This option is useful when execution takes significant time, or if you set breakpoints. You can also close dialog manually. When called function finishes execution, OllyDbg will automatically reopen Call export. "Pause after call" means that *loaddll* will be paused after execution.

Press "Call". OllyDbg automatically backups all dumps, verifies and calculates parameters and registers, removes dialog from the screen and then calls *MessageBoxW*. As expected, message box appears on the screen:

{bmc bm2.WMF} 

MessageBoxW doesn't modify its arguments. If you call function that updates memory, like *GetWindowName*, dump will highlight modified bytes. Note that EAX is 1 on return, which means success.

Another example is available on my Internet site, http://home.t-online.de/home/Ollydbg/Loaddll.htm.

Unfortunately, you can't debug OllyDbg plugins in this way. Plugins are linked to *ollydbg.exe*, and Windows is unable to load and run two executable files in one application.

$#K LOADDLL.EXE

This is the full source code of *loaddll.exe*. It can be compiled using Borland's TASM32 with following commands:

tasm32 -mx -zi -m5 loaddll.asm,,loaddll.lst
tlink32 -v- -c -S:40000 -B:400000 -Tpe -aa -m loaddll,,,import32.lib
brc32 loaddll.rc -feloaddll.exe

Execution begins at **START**. *loaddll* gets command line, skips name of executable (must be taken into double quotes!), extracts path to DLL and passes it to *LoadLibrary*. On error, it places pointer to error message on fixed location and exits with code 0x1001. On success, it creates simple main window and pauses on **Firstbp**. This breakpoint is set by OllyDbg on startup.

All communication with OllyDbg is done through the 128-byte link area. This area must begin at address 0x420020 immediately after keyphrase. First several words contain addresses in *loaddll.exe* used by OllyDbg to set breakpoints and parameters, followed by address of function to call, contents of registers, number of arguments and arguments itself. Number of arguments is limited to 10. If argument is a pointer to memory, you can use 10 data buffers, 1 Kbyte each, named as **Arg1**, **Arg2**, ..., **Arg10**. These and some other names are exported and thus known to OllyDbg.

When loaddll passes main windows loop (**WINLOOP**), it constantly checks whether address of exported function in **PROCADR** is not 0. If this is the case, loaddll saves contents of ESP and EBP and pushes 16 zeros into stack. This is necessary to avoid crash if user specifies invalid number of arguments. Then it pushes arguments and sets registers. At address **Prepatch** there are 16 NOPs that you can use for small patches. If you need more space, you can jump to **Patcharea** 2 Kbytes long. Note that OllyDbg doesn't extract *loaddll.exe* from resources if file with this name already exists.

At **CallDLL** export is called. This command is followed by another 16 NOPs. Then routine saves modified registers and offset of ESP after call. If you supply invalid number of arguments to PASCAL-style function, OllyDbg will be able to report this error to you. Finally, loaddll restores ESP and EBP, zeroes **PROCADR** and breaks at INT3 at address **Finished**. When this point is reached, OllyDbg knows that execution is finished.

Treat *LOADDLL.ASM* as a freeware. I will not protest if you use this program as whole or in parts (without copyright) in your own programs. But do not dare to use the Green Bug (*LOADDLL.RC*) in projects not related to OllyDbg… That's all, enjoy!

LOADDLL.ASM:

```
P586                          ; 32-bit instructions used!
MODEL FLAT,PASCAL
IDEAL                         ; I really like it!
LOCALS

PUBLICDLL WndProc,Firstbp,Prepatch,CallDLL,Finished;
PUBLICDLL Patcharea,Endpatch
PUBLICDLL Arg1,Arg2,Arg3,Arg4,Arg5,Arg6,Arg7,Arg8,Arg9,Arg10

SEGMENT _DATA1 PARA PUBLIC USE32 'DATA'

; Text below is a keyphrase used by OllyDbg to verify that LoadDll
```

```
          ; is correct. Loads at fixed address 00420000. Never change! Note
          ; how coward I am: you cannot replace copyright, otherwise this code
          ; will not work!
          DB "DLL Loader (C) 2004 Oleh Yuschuk"

          ; Link area. Never change the meaning or order of next 32 dwords!
ERRMSG    DD 0                        ; Pointer to error
HINST     DD 0                        ; Process instance
HWND      DD 0                        ; Handle of main window
DLLBASE   DD 0                        ; Base address of loaded DLL or NULL
          DD OFFSET Firstbp           ; Address of first breakpoint
          DD OFFSET Prepatch          ; Address of patch area before call
          DD OFFSET Arg1              ; Base of 10 arguments x 1024 bytes
          DD OFFSET Finished          ; Address of breakpoint after call
DUMMY     DD 4 DUP(0)                 ; Reserved for the future
PROCADR   DD 0                        ; Address of procedure, starts execution
REGEAX    DD 0                        ; Register arguments
REGECX    DD 0
REGEDX    DD 0
REGEBX    DD 0
REGESI    DD 0
REGEDI    DD 0
NARG      DD 0                        ; Number of arguments to push on stack
ARGLIST   DD 10 DUP(0)                ; DLL argument list
ESPDIFF   DD 0                        ; Difference in ESP caused by code
          DD 0                        ; Reserved for the future

WCLASS    = THIS DWORD                ; Hand-made WNDCLASS structure
          DD 0000002Bh                ; CS_HREDRAW|VREDRAW|DBLCLKS|OWNDC
          DD WndProc                  ; Window procedure
          DD 0                        ; Class extra bytes
          DD 0                        ; Window extra bytes
WCINST    DD 0                        ; Instance
WCICON    DD 0                        ; Icon
HCURS     DD 0                        ; Cursor
HBGND     DD 0                        ; Background brush
          DD 0                        ; No menu
          DD CLSNAME                  ; Class name

MSG       = THIS DWORD                ; Hand-made MSG structure
          DD 0                        ; Handle of window
MSGID     DD 0                        ; Message ID
          DD 0                        ; wParam
          DD 0                        ; lParam
          DD 0                        ; Timestamp
          DD 0                        ; X coordinate
          DD 0                        ; Y coordinate

PSTRUCT   = THIS DWORD                ; Hand-made PAINTSTRUCT structure
          DD 0                        ; HDC
          DD 0                        ; fErase
          DD 0                        ; rcPaint.left
          DD 0                        ; rcPaint.top
          DD 0                        ; rcPaint.right
          DD 0                        ; rcPaint.bottom
          DD 0                        ; fRestore
          DD 0                        ; fIncUpdate
          DB 32 DUP(0)                ; rgbReserved

ORIGESP   DD 0                        ; Original ESP before call
ORIGEBP   DD 0                        ; Original EBP before call
EXPESP    DD 0                        ; Expected ESP after call (C)
WNDNAME   DB "OllyDbg DLL Loader",0
```

```
CLSNAME     DB "LoadDLLClass",0
ICONAME     DB "MAINICON",0             ; Green smashed bug - igitt!
E_NONAM     DB "Missing DLL name",0     ; Error notifications to OllyDbg
E_NODLL     DB "Unable to load DLL",0
E_NPARM     DB "Too many parameters",0

            ALIGN 16
Arg1        DB 1024 DUP (?)             ; Area for 10 memory arguments, 1 k each
Arg2        DB 1024 DUP (?)
Arg3        DB 1024 DUP (?)
Arg4        DB 1024 DUP (?)
Arg5        DB 1024 DUP (?)
Arg6        DB 1024 DUP (?)
Arg7        DB 1024 DUP (?)
Arg8        DB 1024 DUP (?)
Arg9        DB 1024 DUP (?)
Arg10       DB 1024 DUP (?)

            ENDS _DATA1

            SEGMENT _TEXT1 PARA PUBLIC USE32 'CODE'

            EXTRN GetModuleHandleA:     PROC
            EXTRN GetCommandLineA:      PROC
            EXTRN LoadIconA:            PROC
            EXTRN LoadCursorA:          PROC
            EXTRN GetStockObject:       PROC
            EXTRN RegisterClassA:       PROC
            EXTRN CreateWindowExA:      PROC
            EXTRN DestroyWindow:        PROC
            EXTRN PostQuitMessage:      PROC
            EXTRN ShowWindow:           PROC
            EXTRN Sleep:                PROC
            EXTRN BeginPaint:           PROC
            EXTRN EndPaint:             PROC
            EXTRN DefWindowProcA:       PROC
            EXTRN LoadLibraryA:         PROC
            EXTRN PeekMessageA:         PROC
            EXTRN TranslateMessage:     PROC
            EXTRN DispatchMessageA:     PROC
            EXTRN ExitProcess:          PROC

            ; Window procedure of main LoadDLL window.
            PROC WndProc
            ARG LP:DWORD,WP:DWORD,MS:DWORD,HW:DWORD
            PUSH EDX
            PUSH EDI
            PUSH ESI
            MOV EAX,[MS]
            CMP EAX,0001h               ; WM_CREATE
            JE RET0
            CMP EAX,0002h               ; WM_DESTROY
            JNE @@080
              PUSH 0
              CALL PostQuitMessage
              JMP RET0
@@080:      CMP EAX,000Fh               ; WM_PAINT
            JNE @@100
              PUSH OFFSET PSTRUCT
              PUSH [HW]
              CALL BeginPaint
              PUSH OFFSET PSTRUCT
              PUSH [HW]
```

```
            CALL EndPaint
            JMP RET0
@@100:      CMP EAX,0010h               ; WM_CLOSE
            JNE @@200
              PUSH [HW]
              CALL DestroyWindow
              JMP RET0
@@200:      ; None of listed above, pass to DefWindowProc().
            PUSH [LP]
            PUSH [WP]
            PUSH [MS]
            PUSH [HW]
            CALL DefWindowProcA
            JMP RETA
RET0:       XOR EAX,EAX
            JMP SHORT RETA
RET1:       MOV EAX,1
RETA:       POP ESI
            POP EDI
            POP EDX
            RET
            ENDP WndProc

START:      MOV EBP,ESP                 ; Here execution begins
            PUSH 0
            CALL GetModuleHandleA
            MOV [DWORD DS:WCINST],EAX
            MOV [DWORD DS:HINST],EAX
            CALL GetCommandLineA        ; Path to LOADDLL is taken into quotes
            MOV ESI,EAX
            INC ESI                     ; Skip first quote
@@10:         MOV AL,[BYTE DS:ESI]      ; Skip path to LOADDLL.EXE
              INC ESI
              OR AL,AL
              JNE @@12
                MOV [DWORD DS:ERRMSG],OFFSET E_NONAM
                JMP ERROR
@@12:         CMP AL,'"'
              JNE @@10
@@20:         MOV AL,[BYTE DS:ESI]      ; Skip spaces
              CMP AL,' '
              JNE @@30
              INC ESI
              JMP SHORT @@20
@@30:       PUSH ESI
            CALL LoadLibraryA           ; Load DLL
            OR EAX,EAX
            JNE @@32
              MOV [DWORD DS:ERRMSG],OFFSET E_NODLL
              JMP ERROR
@@32:       MOV [DWORD DS:DLLBASE],EAX
            PUSH OFFSET ICONAME
            PUSH [DWORD DS:HINST]
            CALL LoadIconA
            MOV [DWORD DS:WCICON],EAX
            PUSH 7F88h                  ; IDC_NO
            PUSH 0                      ; External resource
            CALL LoadCursorA
            MOV [DWORD DS:HCURS],EAX
            PUSH 0                      ; WHITE_BRUSH
            CALL GetStockObject
            MOV [DWORD DS:HBGND],EAX
            PUSH OFFSET WCLASS
```

```
            CALL RegisterClassA
            PUSH 0                      ; Parameters: none
            PUSH [DWORD DS:HINST]       ; Instance
            PUSH 0                      ; Menu: none
            PUSH 0                      ; Parent window: none
            PUSH 100                    ; Width
            PUSH 200                    ; Height
            PUSH 80000000h              ; CW_USEDEFAULT
            PUSH 80000000h              ; CW_USEDEFAULT
            PUSH 10CF0000h              ; WS_OVERLAPPEDWINDOW|WS_VISIBLE
            PUSH OFFSET WNDNAME         ; Window name
            PUSH OFFSET CLSNAME         ; Class name
            PUSH 0                      ; Extended style: none
            CALL CreateWindowExA
            MOV [DWORD DS:HWND],EAX     ; Save handle
            PUSH 9                      ; SW_RESTORE
            PUSH EAX
            CALL ShowWindow
Firstbp:    NOP                         ; First breakpoint is set here
WINLOOP:      CMP [DWORD DS:PROCADR],0  ; Request to call some function?
              JE NOCALL
                MOV [DWORD DS:ORIGESP],ESP
                MOV [DWORD DS:ORIGEBP],ESP
                PUSH 0                  ; Security buffer (16 doublewords)
                PUSH 0
                PUSH 0
                PUSH 0
                PUSH 0
                PUSH 0
                PUSH 0
                PUSH 0
                PUSH 0
                PUSH 0
                PUSH 0
                PUSH 0
                PUSH 0
                PUSH 0
                PUSH 0
                MOV ECX,[DWORD DS:NARG]
                JECXZ @@44
                CMP ECX,10
                JBE @@40
                  MOV [DWORD DS:ERRMSG],OFFSET E_NPARM
                  JMP ERROR
@@40:           MOV EAX,OFFSET ARGLIST
@@42:             PUSH [DWORD EAX]      ; Push requested number of arguments
                  ADD EAX,4
                  LOOP @@42
@@44:           MOV [DWORD DS:EXPESP],ESP ; Expected ESP after return (C)
                MOV EAX,[DWORD DS:REGEAX] ; Preset registers
                MOV ECX,[DWORD DS:REGECX]
                MOV EDX,[DWORD DS:REGEDX]
                MOV EBX,[DWORD DS:REGEBX]
                MOV ESI,[DWORD DS:REGESI]
                MOV EDI,[DWORD DS:REGEDI]
Prepatch:       NOP                     ; Patch area before call
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
```

```
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
CallDLL:        CALL [DWORD DS:PROCADR] ; Call DLL function
                NOP                     ; Patch area after call
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                NOP
                MOV [DWORD DS:REGEAX],EAX ; Get modified registers
                MOV [DWORD DS:REGECX],ECX
                MOV [DWORD DS:REGEDX],EDX
                MOV [DWORD DS:REGEBX],EBX
                MOV [DWORD DS:REGESI],ESI
                MOV [DWORD DS:REGEDI],EDI
                MOV EAX,ESP
                SUB EAX,[DWORD DS:EXPESP]
                MOV [DWORD DS:ESPDIFF],EAX
                MOV EBP,[DWORD DS:ORIGEBP]
                MOV ESP,[DWORD DS:ORIGESP]
                MOV [DWORD DS:PROCADR],0 ; Confirm execution
                NOP
Finished:       INT 3                   ; Pause after execution
                NOP
NOCALL:       PUSH 0
              CALL Sleep                ; Be fair to other applications
              PUSH 1                    ; PM_REMOVE
              PUSH 0                    ; Process all messages
              PUSH 0
              PUSH 0                    ; Any window
              PUSH OFFSET MSG
              CALL PeekMessageA
              OR EAX,EAX
              JZ WINLOOP
              PUSH OFFSET MSG
              CALL TranslateMessage
              PUSH OFFSET MSG
              CALL DispatchMessageA
              MOV EAX,[DWORD DS:MSGID]
              CMP EAX,12h               ; WM_QUIT
              JNE WINLOOP
            PUSH 0
            CALL ExitProcess            ; Hasta la vista!
ERROR:      PUSH 00001001h              ; Special return code, means error
            CALL ExitProcess            ; Error detected
            ALIGN 4
```

```
Patcharea: DB 2047 DUP(90h)            ; Big patch area (2 K of NOPs)
Endpatch:  NOP
           ENDS _TEXT1

           END START
```

LOADDLL.RC:

```
MAINICON ICON                          // Green bug
{
 '00 00 01 00 02 00 20 20 10 00 00 00 00 00 E8 02'
 '00 00 26 00 00 00 10 10 10 00 00 00 00 00 28 01'
 '00 00 0E 03 00 00 28 00 00 00 20 00 00 00 40 00'
 '00 00 01 00 04 00 00 00 00 00 80 02 00 00 00 00'
 '00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'
 '00 00 00 00 80 00 00 80 00 00 00 00 80 80 00 80 00'
 '00 00 80 00 80 00 80 80 00 00 80 80 80 00 C0 C0'
 'C0 00 00 00 FF 00 00 FF 00 00 00 FF FF 00 FF 00'
 '00 00 FF 00 FF 00 FF FF 00 00 FF FF FF 00 00 00'
 '00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'
 '00 00 00 00 02 A0 00 00 00 00 A2 00 00 00 00 00'
 '00 00 00 00 0A 20 00 00 00 0A 2A 2A 00 00 00 00'
 '00 00 00 00 A2 A2 00 00 00 A2 A2 A2 00 00 00 00'
 '00 00 00 00 2A 2A 00 00 0A 2A 2A 2A 00 00 00 00'
 '00 00 00 00 A2 A2 00 00 A2 A2 A2 A0 00 00 00 00'
 '00 00 00 00 2A 2A 00 0A 2A 2A 2A 20 00 00 02 A2'
 '00 00 00 00 02 A2 A0 A2 A2 A2 A2 00 00 00 0A 2A'
 '2A 2A 00 00 0A 2A 2A 2A 2A 2A 20 00 00 00 02 A2'
 'A2 A2 A0 00 02 A2 A2 A2 A2 A2 00 00 00 00 0A 2A'
 '2A 2A 2A 2A 2A 2A 2A 2A 2A 00 00 00 00 00 00 A2'
 'A2 A2 A2 A2 A2 A2 A2 A2 00 00 00 00 00 00 00 00'
 '2A 2A 2A 2A 2A 2A 2A 2A 00 00 00 00 00 00 00 00'
 '00 02 A2 A2 A2 A2 A2 A2 00 00 00 00 00 00 00 00'
 '00 00 0A 2A 2A 2A 2A 2A 2A 00 00 00 00 00 00 00'
 '00 00 02 A2 A2 A2 A2 A2 A2 A2 A2 00 00 00 00 00'
 '00 00 0A 2A 2A 2A 2A 2A 2A 2A 2A 20 00 00 00 00'
 '00 00 02 A2 A2 A2 A2 A2 A2 A2 A2 A2 A2 00 00 00'
 '00 00 0A 2A 2A 2A 2A 2A 00 00 2A 2A 2A 20 00 00'
 '00 00 A2 A2 A0 A2 A2 A0 00 00 02 A2 A2 A0 00 00'
 '00 00 2A 2A 00 0A 2A 20 00 00 00 2A 2A 20 00 00'
 '00 00 00 00 00 00 A2 A0 00 00 00 00 00 00 00 00'
 '00 00 00 00 00 00 2A 20 00 00 00 00 00 00 00 00'
 '00 00 00 00 00 00 A2 A0 00 00 00 00 00 00 00 00'
 '00 2A 20 00 00 00 2A 2A 00 00 00 00 00 00 00 00'
 'A2 A2 A0 00 00 00 A2 A2 00 00 00 00 00 00 00 00'
 '2A 2A 20 00 00 00 2A 2A 00 00 00 00 00 00 00 00'
 'A2 A2 A0 00 00 00 02 A2 00 00 00 00 00 00 00 0A'
 '2A 2A 20 00 00 00 0A 2A 00 00 00 00 00 00 00 02'
 'A2 A2 00 00 00 00 00 00 00 00 00 00 00 00 00 0A'
 '2A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00'
 '00 00 00 00 00 00 00 00 00 00 00 00 00 00 FF F9'
 'FF 3F FF F0 FE 0F FF F0 FC 07 FF E0 78 07 FF E0'
 '70 07 FF E0 60 0F 8F E0 40 0F 00 F0 00 1F 00 70'
 '00 3F 00 10 00 7F 00 00 00 FF 80 00 03 FF C0 00'
 '0F FF E0 00 03 FF FC 00 00 3F FF 00 00 1F FF 00'
 '00 03 FF 00 00 01 FF 00 00 00 FE 00 0F 00 FE 04'
 '0F 80 FF 0E 0F C1 FF FE 0F FF FC 7E 0F FF F0 3E'
 '07 FF E0 3E 07 FF E0 3E 07 FF C0 3F 07 FF C0 3F'
 '07 FF C0 7F 0F FF C0 FF FF FF E3 FF FF FF 28 00'
 '00 00 10 00 00 00 20 00 00 00 01 00 04 00 00 00'
 '00 00 C0 00 00 00 00 00 00 00 00 00 00 00 00 00'
 '00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 80'
 '00 00 00 80 80 00 80 00 00 00 80 00 80 00 80 80'
```

```
 '00 00 80 80 80 00 C0 C0 C0 00 00 00 FF 00 00 FF'
 '00 00 00 FF FF 00 FF 00 00 00 FF 00 FF 00 FF FF'
 '00 00 FF FF FF 00 00 00 00 00 00 00 00 00 00 00'
 '00 AA 00 0A AA 00 00 00 00 AA 00 AA AA 00 00 00'
 '00 AA 0A AA A0 00 0A AA 00 0A AA AA 00 00 0A AA'
 'AA AA AA 00 00 00 00 0A AA AA AA 00 00 00 00 00'
 '0A AA AA A0 00 00 00 00 0A AA AA AA AA 00 00 00'
 'AA AA AA 00 AA A0 00 00 00 00 A0 00 00 00 00 00'
 '00 00 A0 00 00 00 00 0A 00 00 AA 00 00 00 00 AA'
 '00 00 AA 00 00 00 00 AA 00 00 00 00 00 00 00 00'
 '00 00 00 00 00 00 FC E3 00 00 F8 41 00 00 F8 01'
 '00 00 88 03 00 00 00 07 00 00 00 0F 00 00 80 1F'
 '00 00 E0 03 00 00 F0 01 00 00 E0 00 00 00 F0 31'
 '00 00 EE 3F 00 00 86 1F 00 00 86 1F 00 00 87 3F'
 '00 00 8F FF 00 00'
}
```

$#K Analysis

OllyDbg integrates quick yet powerful code analyzer. To launch it, use pop-up menu or press **Ctrl+A** in the Disassembler pane of the CPU window or select "Analyze all modules" in Executable modules.

The Analyzer is highly heuristical. It distinguishes code from data, marks entry points and jump destinations, recognizes switch tables, ASCII and UNICODE strings, locates procedures, loops, high-level switches and decodes arguments of standard API functions (see example). Other parts of OllyDbg make extensive use of analysis data.

How is it possible? I'll open a piece of this Geheimniss to you. On the first pass OllyDbg disassembles every possible address within the code section and 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, I'm sure that this address is entry point to some frequently used subroutine. Starting from located entries, I trace all jumps and calls, 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 around 20 highly efficient heuristical methods (the simplest looks like: "Direct access to first 64 K of memory, like in MOV [0],EAX, is not allowed").

Sometimes analysis fails at the point of your interest. There are two workarounds in this situation: either remove analysis from selection (shortcut **BkSpc**) and OllyDbg will use default decoding (disassembly). Or, set decoding hints and repeat analysis. Note that in some rare cases, when Analyzer is sure that your hint does not apply or is conflicting, it may ignore your suggestion.

Detection of procedures is also simple. Procedure, from the Analyzer's point of view, is 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). You can specify one of three recognition levels. Strict procedure has exactly one entry point and at least one return. On heuristical level, Analyzer tries to assume entry by itself. And 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 split procedure in several parts. In such case fuzzy mode is especially useful. The probability of misinterpretation, however, is rather high.

Similarly, loop is a closed continuous sequence of commands where last command is a jump to the first with a single entry point and any number of exits. Loops correspond to high-level operators *do*, *while* and *for.* OllyDbg recognizes nested loops of any complexity. They are marked by long fat parenthesis in the disassembly. If entry is not the first command in the loop, OllyDbg marks it with a small triangle.

#To implement a switch, many compilers load switch variable into 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 of comparisons only), then this is probably not a switch, but a cascaded if operator:

```
if (i==0) {...}
else if (i==5) {...}
else if (i==10) {...}
```

To let OllyDbg decode cascaded if's as switches, select corresponding option in Analysis1.

OllyDbg contains, as an internal precompiled resource, the descriptions of more than 1900 frequently used API functions. This list includes KERNEL32, GDI32, USER32, ADVAPI32, COMDLG32, SHELL32, VERSION, SHLWAPI, COMCTL32, WINSOCK, WS2_32 and MSVCRT. You can add your own descriptions. If Analyzer encounters call to the function with known name (or call to jump to such function), it tries to decode PUSH commands that immediately precede this call. Consequently, you can interprete the meaning of calls at a glance. OllyDbg has also descriptions of about 400 standard C functions. If you posess original libraries, I recommend that you scan object files before analysis. In this case, OllyDbg will decode arguments of known C functions too.

If option "Guess number of arguments of unknown functions" is set, Analyzer tries to determine number of doublewords that calling procedure pushes onto the stack, and marks them as arguments Arg1, Arg2 and so on. Notice that register arguments, if any, remain unrecognized and are not included in this count. Analyzer goes a secure way. For example, it doesn't recognize procedures without parameters or cases where POP may restore registers before return instead of discarding arguments. Nevertheless, the number of recognized functions is usually very high, strongly improving the readability of code.

#Analyzer can trace the contents of integer registers. Modern optimizing compilers, especially Pentium-oriented, frequently load constants or addresses into registers for repeated use or to minimize memory latencies. If some constant is loaded into register, Analyzer notices it and uses to decode functions and their arguments. It is also able to perform simple arithmetical calculations and even traces pushes and pops.

Analyzer doesn't distinguish between different kinds of names. If you assign known name to some function, OllyDbg will decode all calls to this address as if there were original routine. There are special predefined names WinMain, DllEntryPoint and WinProc. You can use these labels to mark entry to the main program, optional DLL entry point and window procedures (notice that OllyDbg does not check user-defined labels for uniqueness). Another, better way to do this is to assume predefined arguments.

Unfortunately, there are no general rules allowing to perform 100% accurate analysis. In some cases, for example when module contains p-code or large amounts of data embedded in the code section, analyzer may interpret parts of this data as a code. If statistical analysis shows that code is probably packed or encrypted, Analyzer will warn you. If you want to use hit trace, I recommend that you don't use fuzzy analysis, otherwise the chances are high that breakpoint will be set on the misinterpreted data.

Self-extractable files usually have extractor outside the "official" code section. If you select the SFX option "Extend code section to include self-extractor", OllyDbg will, er, extend code section, formally allowing to analyze it and apply hit trace and selective run trace.

See also: Object scanner, Analysis options - part 1, Analysis options - part 2, Analysis options - part 3, Call tree

$#K Decoding hints

In some cases Analyzer is unable to distinguish between code and data. Let's have a look at the following example:

```
const char s[11] = "0123456789";
...
for (i=0x30; i<0x3a; i++) t[i-0x30]=s[i-0x30];
```

Good optimizing compiler will remove subtraction and replace it with something like

```
for (i=0x30; i<0x3a; i++) (t-0x30)[i]=(s-0x30)[i];
```

where `t-0x30` and `s-0x30` are, of course, constants, and compile assignment to

```
MOV AL,[BYTE s_minus_30+EBX]
MOV [BYTE t_minus_30+EBX],AL
```

Compiler may also embed constant string `"0123456789"` into the executable code. In version 1.10, I do not attempt to determine range of register's values. When Analyzer encounters above commands, it assumes that address `s_minus_30` contains byte data. But in reality, it may contain code!

What if such error appeared in the code you are going to debug? There are two possibilities. Quick and dirty: you may delete analysis from the misinterpreted code (shortcut: **BkSpc**) and OllyDbg will use default decoding -disassembly.

More intelligent option are decoding hints. You tell OllyDbg how to interprete contents of selected memory areas and re-run analysis (**Ctrl+A**).

To set hint, select piece of code or data in Disassembler and from the pop-up menu select **Analysis|During next analysis, treat selection as** and then one of available options:

Command - first selected byte starts valid command. This is a sure command, that is, all subsequent locations till jump or return, and all locations reachable by jumps or calls from this sequence are commands, too;

Byte,
Word,
Doubleword - treat first 1, 2 or 4 selected bytes as data of corresponding size;

Commands - the whole selection (till first invalid command) and all reachable destinations consist of valid commands;

Bytes,
Words,
Doublewords - selection consists of 1-, 2- or 4-byte items;

ASCII text,
UNICODE text - the whole selection is ASCII or UNICODE text;

Default (remove hints) - removes all hints from selection;

Remove all hints - removes decoding hints from the whole module.

OllyDbg saves hints to .udd file.

$#K# Object scanner

Object scanner takes specified object files or libraries (both in OMF and COFF formats), extracts code segments and then tries to locate these segments in the code section of the current module. If segment is located, Scanner adds names from the object file to debugging information (the so-called library labels). This greatly improves the understandability of both code and data.

Scanner doesn't try to match labels from the object file with recognized names, so it can't recognize very small or similar procedures (i.e. procedures that differ only in relocations). Always check the list of warnings that scanner writes to log window!

See also: Analysis, Implib scanner

$#K# Implib scanner

Some DLLs export their symbols only by ordinals. Such symbols appear as sharp numbers (like MFC42.#1003) and are usually not very meaningful for human beings. Fortunately, software vendors usually supply import libraries (implibs) which associate ordinals with symbolic names. OllyDbg can use these symbols.

To use implib scanner, select 'Debug|Select import libraries' from the main menu and enter names of requested import libraries. When you load application, OllyDbg reads libraries and extract names to internal tables. Each time it encounters reference to ordinal from the registered IMPLIB, it substitutes it with symbolic name.

See also: Analysis, Object scanner

$#KBreakpoints

OllyDbg supports several kinds of breakpoints:

- **Ordinary breakpoint**, where the first byte of the command you want to break on is replaced by a special command INT3 (Trap to Debugger). You place this breakpoint by selecting the command in Disassembler and pressing **F2**, or over pop-up menu. When you press **F2** for the second time, breakpoint will be removed. Notice that program stops before command with breakpoint is executed.
The number of INT3 breakpoints you may set is unlimited. When you close debugged program or Debugger, OllyDbg automatically saves breakpoints to disk. Never try to set this breakpoint on data or in the middle of the command! OllyDbg warns you if you attempt to set breakpoint outside the code section. You can permanently turn off this warning in Security options. In some cases Debugger may insert own temporary INT3 breakpoints.

- **Conditional breakpoint** (shortcut **Shift+F2**) is an ordinary INT3 breakpoint with associated expression. Each time Debugger encounters this breakpoint, it estimates expression and, if result is non-zero or expression is invalid, stops debugged program. Of course, the overhead caused by false conditional breakpoint is very high (mostly due to latencies of the operational system). On PII/450 under Windows NT OllyDbg processes up to 2500 false conditional breakpoints per second. An important case of conditional breakpoint is break on Windows message (like WM_PAINT). For this purpose you can use pseudovariable MSG together with proper interpretation of arguments. If window is active, consider message breakpoint described below.

#- **Conditional logging breakpoint** (**Shift+F4**) is a conditional breakpoint with the option to log the value of some expression or arguments of known function each time the breakpoint is encountered or when condition is met. For example, you can set logging breakpoint to some window procedure and list all calls to this procedure, or only identifiers of received WM_COMMAND messages, or set it to the call to CreateFile and log names of the files opened for read-only access etc. Logging breakpoints are as fast as conditional, and of course it's much easier to look through several hundred messages in the log window than to press F9 several hundred times. You can select one of several predefined interpretations to your expression.

You may set pass count – a counter that decrements each time the pause condition is met. If pass count before decrement is not zero, OllyDbg will continue execution. Consider a loop that executes 100. (decimal) times. Place breakpoint in the middle and set pass count to 99. (decimal). OllyDbg will break on the last iteration.

Additionally, conditional logging breakpoint allows you to pass one or several commands to plugins. For example, this may be a request for command-line plugin to change contents of register and continue.

- **Message breakpoint** is same as conditional logging except that OllyDbg automatically generates condition allowing to break on some message (like WM_PAINT) on the entry point to window procedure. You can set it in the Windows window.

- **Trace breakpoint** is an extreme case of INT3 breakpoint set on every specified command. If you make hit trace, breakpoint will be removed and address marked as hit after command is reached. If you use run trace, OllyDbg adds record to trace data and breakpoint remains active.

- **Memory breakpoint**. OllyDbg allows single memory breakpoint at a time. You select some portion of memory in Disassembler or CPU Dump and use pop-up menu to set memory breakpoint. Previous memory breakpoint, if any, will be automatically removed. You have two options: break on memory access (read, write or execute) and on write only. To set this breakpoint, OllyDbg changes attributes of memory blocks containing selection. On

80x86-compatible processors memory is allocated and protected in chunks of 4096 bytes. If you select even single byte, OllyDbg must protect the whole block. This may cause plenty of false alarms with huge overhead. Use this kind of breakpoint with care. Some system functions (especially under Windows 95/98) cause debugged program to crash instead of generating debugging event when accessing protected memory.

- **Hardware breakpoint** (available only when running Debugger under Windows ME, NT, 2000 or XP). 80x86-compatible processors allow you to set 4 hardware breakpoints. Unlike memory breakpoint, hardware breakpoints do not slow down the execution speed, but cover only up to 4 bytes. OllyDbg can use hardware breakpoints instead of INT3 when stepping or tracing through the code.

#- **Single-shot break on memory access** (available only under Windows NT, 2000 and XP). You set it in Memory window on the whole memory block from the pop-up menu or by pressing **F2**. This breakpoint is especially useful if you want to catch call or return to some module. After break happens, breakpoint is removed.

- **Run trace pause** (shortcut **Ctrl+T**) is a set of conditions that are checked on each step of the run trace. You can pause run trace if EIP enters some range or leaves another range, or some condition is true, or when command matches one of the specified patterns, or when command is suspicious, or after specified number of commands is traced. Note that this option can significantly (up to 20%) slow down the speed of run trace.

OllyDbg can also stop program execution on some **debugging events**, like loading or unloading DLL, starting or terminating thread, or when program emits debug string.

$#K# Shortcuts

$#K# Global shortcuts

These shortcuts are OllyDbg-wide, dependless on the currently active window:

Ctrl+F2 - program reset, starts over debugged program. If there is no active program, OllyDbg restarts first program in the history list. Program reset removes memory and hardware breakpoints.

Alt+F2 - close, closes debugged program. If program is still active, you will be asked to confirm the action.

F3 - displays „Open 32-bit .EXE file" dialog box where you can select executable file and optionally specify arguments.

Alt+F5 - makes OllyDbg topmost. If debugged program stops on breakpoint while displaying topmost window (usually some kind of modal message or dialog), this window may cover parts of OllyDbg but you are unable to move or minimize it without continuation. Activate OllyDbg (for example, from the taskbar) and press Alt+F5. OllyDbg will get topmost and place itself over the Debuggee. If you press Alt+F5 for the second time, OllyDbg becomes normal (non-topmost) window. Topmost status is preserved between debugging sessions. Actual OllyDbg status is displayed in the status bar.

F7 - step into, executes next single command. If this command is a function call, stops on the call destination. If command has REP prefix, executes single iteration of the command.

Shift+F7 - same as F7, but if debugged program stopped on some exception, Debugger will first try to pass exception to handler specified in the debugged program (see also Ignore memory access violations in Kernel32).

Ctrl+F7 – animate into, executes commands step-by-step, also entering function calls (as if you press and hold F7, only faster). Animation stops when you execute some other stepping or continuation command, program reaches active breakpoint or some exception happens. Each time next step is executed, OllyDbg redraws all windows. To speed up animation, close all windows you don't use and reduce the size of remaining windows. To stop animation, press Esc.

F8 - step over, executes next single command. If this command is a function call, executes called function at once (except when function contains breakpoint or produces exception). If command has REP prefix, executes all iterations and stops on the next command.

Shift+F8 - same as F8, but if debugged program stopped on some exception, Debugger will first try to pass exception to handler specified in the debugged program (see also Ignore memory access violations in Kernel32).

Ctrl+F8 – animate over, executes commands step-by-step, without entering function calls (as if you press and hold F8, only faster). Animation stops when you execute some other stepping or continuation command, program reaches active breakpoint or some exception happens. Each time next step is executed, OllyDbg redraws all windows. To speed up animation, close all windows you don't use and reduce the size of remaining windows. Press Esc to stop animation.

F9 - continues program execution.

Shift+F9 - same as F9, but if debugged program stopped on some exception, Debugger will first try to pass exception to handler specified in the debugged program (see also Ignore

memory access violations in Kernel32).

Ctrl+F9 - execute till return, traces program without entering function calls or updating CPU till the next encountered return. As program is executed step-by-step, this may take some time. Press Esc to stop tracing.

Alt+F9 - execute till user code, traces program without entering function calls or updating CPU and stops when next encountered command belongs to module that doesn't reside in system directory. As program is executed step-by-step, this may take some time. Press Esc to stop tracing.

Ctrl+F11 - run trace into, executes commands step-by-step entering function calls and adding contents of registers to run trace data. Run trace doesn't animate CPU window.

F12 - stops program execution by suspending all threads of debugged program. Don't resume threads manually, rather use ordinary continuation keys and menu items (like F9).

Ctrl+F12 - run trace over, executes commands step-by-step without entering function calls and adds contents of registers to run trace data. Run trace doesn't animate CPU window.

Esc - if animation or tracing is active, stops animation or tracing. If CPU displays trace data, shows actual data.

Alt+B - opens or restores Breakpoints window. Here you can edit, delete or follow breakpoints.

Alt+C - opens or restores CPU window.

Alt+E - opens or restores list of modules.

Alt+K - opens or restores Call stack window.

Alt+L - opens or restores Log window.

Alt+M - opens or restores Memory window.

Alt+O - opens Options dialog.

Ctrl+P - opens Patches window

Ctrl+T - opens Pause run trace dialog

Alt+X - terminates OllyDbg.

Most windows understand following keyboard commands:

Alt+F3 - closes active window.

Ctrl+F4 - closes active window.

F5 - maximizes active window or restores it to normal size.

F6 - activates next window.

Shift+F6 - activates previous window.

F10 - opens pop-up menu associated with acive window or pane.

LeftArrow - shifts contents 1 character to the left

Ctrl+LeftArrow - shifts contents 1 column to the left

RightArrow - shifts contents 1 character to the right

Ctrl+RightArrow - shifts contents 1 column to the right

$#KCPU window

For you as a user, CPU window is the most important in OllyDbg. Here you will spend most of time debugging your application. It consists of five resizable panes:

- Disassembler
- Information
- Dump
- Registers
- Stack

Press **TAB** to activate next CPU pane (in clockwise direction).

Press **Shift+TAB** to activate previous CPU pane (in counterclockwise direction).

See also: Context-sensitive API help.

$#K# Context-sensitive API help

You can attach a help file with descriptions of API functions to OllyDbg and get instant context-sensitive help (shortcut: **Ctrl+F1**) in Disassembler, Names and Intermodular calls windows.

To attach, choose Help|Select API help file from the main menu. Help file (*win32.hlp*) is not included into the distribution *odbg110.zip*.

$#K# Disassembler window

The Disassembler pane of CPU window (I will call it Disassembler window for brevity) displays code of debugged program. It has four columns: address, hex dump, disassembled commands and comments. The last column can alternatively display related source code or run profile. If bar is visible, press it to switch between display modes.

Doubleclicking in some column performs one of the frequently used actions:

Address - displays addresses relative to the clicked. Doubleclick base address again to return to the standard address mode;

Hex dump - toggles unconditional breakpoint;

Disassembly - invokes Assembler;

Comment - allows you to add or edit comment associated with the command.

If Disassembler displays bar on the top of the pane and backup is available, press Address/Backup to toggle display mode between backup and normal view. Press bar over the fourth column to toggle between comment, source code and profile data.

When program pauses, Disassembler scrolls to the command pointed to by the EIP of the current thread.

Disassembler allows you to browse, analyze, search and modify code, copy changes to executable file, set breakpoints and so on. Associated popup menu contains more than 150 entries. As in other cases, menu displays only items that apply to current selection. If several commands are selected, single-command menu items apply to the first selected command.

To select several lines from the keyboard, hold **Shift** key and use **Up**/**Down** arrows or **PgUp**/**PgDn** keys.

To scroll Disassembler bytewise, hold **Ctrl** key and press **Up**/**Down** arrows.

To get more information about Disassembler, click on one of the following topics:

Disassembler menu
Disassembler shortcuts
Disassembler options
Analysis
Breakpoints

See also: Information window, Dump

$#K Step-by-step execution and animation

You step through the debugged program by pressing **F7** (step into) or **F8** (step over). The main difference between these stepping methods is that if current command is a call to some subroutine, **F7** will enter subroutine and stop on its first command, whereas **F8** will try to execute the subroutine at once. If you step over some subroutine, any breakpoint or debugging event within the subroutine will pause execution, but temporary breakpoint after call will remain active and you will hit it sooner or later.

If debugged program stops on exception, you can pass this exception to the handler installed by debugged program. Simply press **Shift** key together with any stepping command.

#Instead of pressing **F7** or **F8** several hundred times, you can use animation (**Ctrl+F7** or **Ctrl+F8**). In this case, OllyDbg automatically repeats **F7** or **F8** after previous step is finished and all windows are updated. The process stops when

- you press **Esc** or issue any other stepping command, or

- OllyDbg encounters breakpoint, or

- debugged program generates exception.

Using **Minus** and **Plus** keys, you can then backtrace the execution history.

Note that OllyDbg redraws most of windows each time the execution is paused. If animation appears to be very slow, try to close or at least minimize all unused windows.

Another, much faster way to backtrace the execution is the run trace. It creates execution protocol and informs you when and how many times given command was executed.

See also: Hit trace, Execute till return

$#K# Execute till return

You may want to skip some routine and return to caller. OllyDbg can trace program execution over all encountered calls and stop on the first encountered return (RET, RETF or IRET). From the main menu, select 'Debug|Execute till return' or press **Ctrl+F9**. Step-by step execution is relatively slow, only up to 5000 commands per second on Pentium/500 under Windows NT. If tracing takes too much time, press **Esc** or **F12** to stop execution till return.

If you want to execute return command and pause program on the next command after corresponding call, activate option "After execution till RET, step over RET" in Trace options.

See also: Execute till user code, Execute till condition

$#K# Execute till user code

Sometimes your application stops somewhere deep in the system code. To return back to the program you are debugging, you can press **Ctrl+F9** (Execute till return) several times, each time checking where you are. There is, however, another, faster way of doing this. If you press **Alt+F9** or select "Debug|Execute till user code" from the main menu, OllyDbg will step over the code as long as it reaches the first command that is not in system DLL. (From the OllyDbg's point of view, any DLL that resides in system or Windows directory is system). Step-by step execution is relatively slow, only up to 5000 commands per second on Pentium/500 under Windows NT. If tracing takes too much time, press **Esc** or **F12** to stop execution till user code.

Caveat: if system DLL calls user callback function, it will be stepped over. To avoid this, you can set single-shot break on access on the user code (available only for Windows NT 4.0, 2000 and XP).

See also: Execute till condition

$#K# Hit trace

Hit trace gives you the possibility to check which parts of the code were executed and which not. The method implemented in OllyDbg is rather straightforward. It sets INT3 breakpoint on every command within the specified region. When breakpoint executes, OllyDbg removes it and marks command as hit. As each trace breakpoint executes only once, this method is very fast.

When using hit trace, special care must be taken not to set breakpoint on data, or with high probability application will crash. For this reason, you *must* analyze code to enable corresponding menu options. I recommend that you select strict or heuristical procedure recognition. Fuzzy option is too error-tolerant and often finds non-existing procedures.

When you set trace breakpoint even on a single command within the module, OllyDbg allocates trace buffers of twice the size of code section.

Note that when you remove hit trace, you simultaneously remove forced run trace.

See also: Disassembler menu, Run trace, Trace options

$#K Run trace

Run trace is the way to backtrace program execution that precedes some event. You can also use run trace for simple profiling. Basically, OllyDbg executes debugged program step-by-step, like in animation, but it doesn't redraw windows and - most important - logs addresses, contents of registers, messages and known operands to the run trace buffer. If debugged code is self-modified, you can save original commands. Start run trace by pressing **Ctrl+F11** (run trace into, entering subroutines) or **Ctrl+F12** (run trace over, executing calls at once), and stop it with **F12** or **Esc**.

#You can specify a set of conditions that are checked on each step of the run trace (shortcut: **Ctrl+T**). Run trace stops if *any* condition is met. Conditions include:

- Pause when EIP is in the address range;
- Pause when EIP is outside the address range;
- Pause when some condition is true;
- Pause when next command is suspicious, i.e. potentially invalid (according to the rules set in Analysis 3), accesses non-existing memory, sets single-step trap flag or accesses stack beyond the actual ESP. Notice that this option can significantly (up to 20%) slow down the speed of run trace;
- Pause after specified number of commands is traced (more precisely, added to run trace buffer). Note that counter doesn't autorestart. That is, if you set command count to 10, trace will pause once after 10 commands are executed, and not on every 10th command.
- Pause when next command matches one of the specified patterns. You can use imprecise commands and operands and matching 32-bit registers RA and RB. Like R32, they substitute any general-purpose 32-bit register but have the same values within the command. RA and RB in this case must be different. For example, in program that consists of XOR EAX,EAX; XOR ESI,EDX, pattern XOR R32,R32 matches both commands; XOR RA,RA matches only the first one, and pattern XOR RA,RB matches only XOR ESI,EDX.

Of course, run trace requires plenty of memory, in average 16 to 35 bytes per command depending on the mode, and is very slow. On a 500-MHz processor under Windows NT it can trace up to 5000 commands per second. Windows 95 is slower: only 2200 commands per second. But in many cases, for example when program jumps to the non-existing address, this is the only way to find the reason. You can exclude quasi-linear sequences of commands (with sole exit at the end of the sequence) from the run trace. When OllyDbg encounters excluded sequence, it sets temporary breakpoint at the command that immediately follows excluded block and runs it at once. Of course, any return or jump to outside would make correct tracing impossible, so OllyDbg checks the piece of code you want to exclude and in hard cases asks you for confirmation.

In most cases you are not interested in tracing system API code. Trace option "Always trace over system DLLs" allows you to trace over API functions in trace/animation into modes. OllyDbg assumes that module is system if it resides in system folder. From the Modules window you can mark any DLL as system or non-system.

To make execution faster, you can limit run trace to selected commands or pieces of code by setting run trace breakpoints and running program. I call this "forced run trace". Basically, run trace breakpoints are non-removable hit trace breakpoints. If you delete hit trace, you simultaneously remove run trace.

Trace commands mentioned at the beginning of this topic automatically open trace buffer. You can specify its size (up to 64 MB) in Options. This buffer is circular and, when full, discards the oldest records.

You can explicitly open or clear run trace buffer by selecting 'Debug|Open or clear run trace' from the main OllyDbg menu. After run trace buffer is open, OllyDbg logs all pauses in execution, even those that were not caused by run trace. For example, you can step through the program by pressing **F7** or **F8** and then backtrace the execution using keys **Plus** and **Minus**. Notice that these keys browse through the history when run trace buffer is closed. If you step through the run trace, Registers and Information panes get grayed to emphasize that registers they display are not actual. Trace buffer doesn't save top of stack or contents of memory referenced by registers. *Registers, Information and Stack use actual memory state* to interprete registers from the run trace.

#OllyDbg can count how many times every command appears in the run trace buffer. In the Disassembler window, select 'View|Profile data'. This command replaces Comments column by Profile. Or, if bar is displayed, click it several times until it displayes Profile information. Notice that displayed count is dynamical and doesn't account for the old commands discarded from the trace buffer. You can also view profile data for the whole module, sorted by hit count, in a separate Profile window.

Special Disassembler command 'Run trace|Add entries of all procedures' allows to check how frequently each recognized procedure is called. Another command 'Run trace|Add branches in procedure' forces trace on all recognized jump destinations within the procedure. In this case, profile allows to find the most frequently executed branch and optimize it for speed.

Option 'Search for|Last record in run trace' in Disassembler pop-up menu finds whether and when given command was executed for the last time.

Run trace window displays the contents of trace buffer. For each command, there is the contents of integer registers that were changed by the command (more precisely, changed from given record to next). If you doubleclick some command, window selects all occurences of the command in the trace buffer and you can quickly browse them by pressing **Plus** and **Minus**. If option 'Trace|Synchronize CPU and Run trace' is set, Disassembler will follow Run trace window.

Notice that when you remove hit trace, you simultaneously remove forced run trace.

See also: Disassembler menu, Hit trace, Trace options

$#K Self-extracting (SFX) files

Self-extracting file consists of extracting routine and packed original program. When troubleshooting SFX, you usually want to skip extractor and stop on the entry point of original program ("real entry"). OllyDbg contains several functions that facilitate this task.

Usually extractor loads to address that is outside the executable section of the original program. In this case OllyDbg recognizes file as SFX.

When SFX options request tracing of real entry, OllyDbg sets memory breakpoint on the whole code section. Initialy this is empty or contains compressed data. When program attempts to execute some command within protected area which is neither RET nor JMP, OllyDbg reports real entry. This is how bytewise extraction works.

This method is very slow. There is another, much faster method. Each time exception on data read occurs, OllyDbg enables reading from this 4-K memory block and disables previous read window. On each data write exception it enables writing to this block and disables previous write window. When program executes command within remaining protected area, OllyDbg reports real entry. However, when real entry is inside read or write window, its location will be reported incorrectly.

You can correct entry position. Select new entry and from Disassembler popup menu choose 'Breakpoint|Set real SFX entry here'. If corresponding SFX option is enabled, next time OllyDbg skips extractor quickly and reliably.

Notice that OllyDbg usually fails to trace extracting routine that implements protection or anti-debugging techniques.

$#KInformation window

Information pane of CPU window decodes arguments of the first command selected in Disassembler pane. Information displays also implicit arguments, like AL and ECX in REPE SCASB , or top of stack in RETN. For analyzed code, it displays list of commands that jump to the current location. If debugging information is available, it also displays source line corresponding to the selection. From here, you can follow addresses in Dump or Disassembler and modify arguments, or open source file.

Any program uses registers very intensively, so their contents continually vary. By default, OllyDbg assumes that registers are valid only for the currently executed command. If you want to decode arguments for all commands, please activate the option Decode registers for any IP.

If you backtrace the run trace log, Information pane gets grayed to emphasize that displayed registers are not actual but taken from the trace data. Notice that run trace does not save contents of memory, all displayed memory contents is actual.

Depending on the selection, pop-up menu of information window may include following items:

Copy pane to clipboard - copies all non-empty lines to clipboard.

Modify register - allows you to change the contents of the selected register.

Follow address in Disassembler - follows address in Disassembler pane.

Follow value in Disassembler - follows contents of doubleword memory pointed to by selected address in Disassembler pane.

Follow address in Dump- follows address in Dump pane of CPU window.

Follow value in Dump - follows contents of doubleword memory pointed to by selected address in Dump pane of CPU window.

Go to CALL from xxxx,
Go to JMP from xxxx,
Go to JNZ from xxxx,
Go to JMP [ ] from xxxx etc. - goes to command that jumps to or calls current location. This can be direct or indirect local (intramodular) call, direct unconditional jump, conditional jump or table switch. Notice that calls from different modules are not listed, even if they are present in call tree.

Show all jumps and local calls - if number of jumps to the current location exceeds 16, this menu item opens dialog that lists all found jumps.

Show source - opens source file at the position corresponding to selected command. Alternatively, doubleclick source line in the Information window.

Appearance - see detailed description here.

$#K Registers window

Registers window displays and interpretes the contents of CPU registers for currently selected thread. It also allows to modify registers and follow addresses in other CPU panes. Pop-up menus associated with each register are self-explanatory, and I will not describe them here.

Following EFL are the suffixes of conditional commands that satisfy current flags. If, for example, you see:

```
EFL 00000A86 (O,NB,NE,A,S,PE,GE,G),
```

this means that JO and JNE will be taken whereas JB and JPO not.

In the same manner, following FST is the decoding of FPU flags C0, C2 and C3 as if the last FPU operation were a comparison.

When debugged program pauses execution, OllyDbg highlights all differences since previous pause. If you modify any item, OllyDbg highlights all modified items.

To toggle value of some flag, doubleclick it, or select and press return. Doubleclicking on register invokes dialog where you can modify its contents. To change integer or FPU registers, start typing new contents. Keys **Plus** (+) and **Minus** (-) increment and decrement selected integer registers.

You can choose either floating-point, MMX or 3DNow! decoding of FPU registers. If window contains bar, pressing the bar will loop through these three formats. You can also let OllyDbg change this presentation automatically, based on the type of the command where last break occured. Alternatively, register window can display debug registers DR0..DR3, DR6 and DR7. However, you can't modify debug registers.

If you are debugging SSE code, activate option "Decode SSE registers" to view 128-bit SSE registers. Notice that this option is slow and sometimes dangerous (especially when application is multithread), don't use it unless absolutely necessary.

If you backtrace the run trace log, Registers window gets grayed to emphasize that displayed registers are not actual but taken from the trace data. Run trace does not save contents of memory, SSE and debug registers and reads actual memory of debugged process to decode memory pointers.

You can display last error detected by a thread (as returned by call to API function *GetLastError*). This feature is controlled by option "Show last error". If activated, last error is saved to run race log, but may significantly (up to 20%) slow down the tracing speed.

You can scroll regiser pane by pressing left mouse button somewhere in the window and moving mouse.

$#K Disassembler shortcuts

OllyDbg recognizes these shortcuts when Disassembler pane of CPU window is active:

ENTER - add selected command to the command history and, if current command is a jump, call or part of the switch table, follow address of destination.

BkSpc - remove analysis from the selection, useful if Analyzer recognized code as data. See also decoding hints.

Alt+BkSpc - undo selection, substitutes selected part of the code with the corresponding portion of backup data. Available only when backup data exists and differs from selected code.

Ctrl+F1 - if API help file is selected, open help topic associated with symbolic name in the first selected line.

F2 - toggle INT3 breakpoint on the first selected command. Alternatively, you can doubleclick line in the second column.

Shift+F2 - set conditional breakpoint on the first selected command. See also Ignore memory access violations in Kernel32.

F4 - run to selection, set one-shot breakpoint on the first selected command and continue execution of debugged program. If OllyDbg catches exception or stops on breakpoint before program reached this command, one-shot breakpoint remains active. If necessary, you can remove it in Breakpoints window.

Shift+F4 - set logging breakpoint (conditional breakpoint with optional logging of value of some expression when condition is met). For more details, see Breakpoints.

Ctrl+F5 - open source file that corresponds to the first selected command.

Alt+F7 - go to the the previous found reference.

Alt+F8 - go to the next found reference.

Ctrl+A - analyse code section of current module.

Ctrl+B - start binary search.

Ctrl+C - copy selection to the clipboard. Copy roughly preserves width of the columns and truncates invisible characters. To exclude some column from the copy, reduce it width to the minimum.

Ctrl+E - edit selection in binary (hexadecimal) format.

Ctrl+F - start command search.

Ctrl+G - go to address. Invokes window asking you to enter address or expression to follow. This command does not modify EIP.

Ctrl+J - list all calls and/or jumps to the current location. You must analyze code before you can use this feature.

Ctrl+K - view Call tree associated with current procedure. You must analyze code before you can

use this feature.

Ctrl+L - search next, repeats last search.

Ctrl+N - open list of names (labels) in current module.

Ctrl+O - scan object files. This command displays the Scan object files dialog where you can select object files or libraries and scan them in an attempt to find object modules within the actual code section.

Ctrl+R - find references to selected command. This command scans through the executable code of the active module and finds all references (constants, jumps or calls) to the first selected command. You can use shortcuts Alt+F7 and Alt+F8 to navigate through the references. For your convenience, referenced command is also included into the list of references.

Ctrl+S - search for command. This command displays Find command dialog where you can enter the assembler command and search for the next instance of this command.

Asterisk (*) - go to the origin (contents of EIP of the active thread).

Ctrl+Gray Asterisk (*) - new origin here, sets EIP of the currently selected thread to the address of the first selected byte. You can undo this operation if you go to Registers pane and select EIP.

Plus (+) - if run trace is inactive, go to the next address from the command history. Otherwise, go to the next record in run trace data.

Ctrl+Plus - go to the beginning of the previous procedure.

Minus (-) - if run trace is inactive, go to the previous address from the command history. Otherwise, go to the next record in run trace data.

Ctrl+Minus - go to the beginning of the next procedure.

Space - assemble. Displays Assemble at dialog where you can edit actual command or enter new commands in Assembler language. These commands substitute actual code. Alternatively, doubleclick the command you are going to change.

Colon (:) - add label. Displays Add label or Change label window where you enter label (symbolic name) associated with the first byte of the first selected command. Notice that in many languages colon is a part of label.

Semicolon (;) - add comment. Displays Add comment or Change comment window where you enter comment (text string displayed in the last column) associated with the first byte of the first selected command. Notice that many Assembler languages use semicolon to start comment. Alternatively, you can doubleclick disassembled line in the Comments column.

$#K Command history

OllyDbg keeps track of up to 1000 last commands displayed in Disassembler window. Each time the debugged program stops on breakpoint or exception, or you step next command, or follow address of jump or call, or simply press Return, OllyDbg adds current address together with thread identifier to the command history. If run trace is inactive, you can navigate through the history by pressing buttons '+' and '-'.

There is also similar data history in CPU Dump.

$#KBackup functions

Disassembler, CPU Dump and all Dump windows can create backup copy of displayed memory block.

Disassembler and CPU Dump create global shared backups. They do it automatically if you modify code or data. Each memory block can have only one global backup. Once created, global backups persist as long as original memory block. Patch manager uses global backups to create list of patches.

Standalone Dump windows create local private backups on your request. When you close Dump window or switch to another memory block, this backup is destroyed. If you have several Dump windows displaying same memory area, their backups are independent.

If backup is available, OllyDbg highlights differences between backup copy and original data. To copy piece of data from backup to original („undo"), select this piece and choose Undo selection from the pop-up menu, or press **Alt+BkSpc**.

One can also write backup to file and load it again. This allows to spy differences between the different runs of debugged program. By choosing Search for|Modified command or Search for|Modified data you can quickly find all differences between backup and original.

Following backup functions are available:

View backup - view backup instead of original data. If window displays backup, all its functionality is disabled. If bar is visible, you can press button „Address" instead.

View actual data - view original data instead of backup. If bar is visible, you can press button „Backup" instead.

Create backup,
Update backup - creates new or updates existing backup.

Delete backup - destroys backup.

Load backup from file - loads backup from file. OllyDbg warns you if size of backup differs from the size of the dump.

Save backup to file - saves backup to file. Default name for memory backup has form MODULE_XXXXXXXX.mem, where MODULE is the short name of the module (or empty if memory block belongs to no module) and XXXXXXXX is the hexadecimal memory base. For file backup, default name is the name of the original file.

Save data to file - saves original data to file.

$#K# Disassembler menu

Disassembler pop-up menu is perharps the most important in the whole OllyDbg. To keep menu compact, it displays only items applicable to the selected part of disassembled code. If several lines are selected at once, single-line commands apply to the first selected line.

Backup functions - see description here.

Copy

To clipboard (Ctrl+C) - copies selected part of the code to clipboard. Uses currently selected width of columns. To exclude some column from the copy, reduce its width to the minimal possible (in this case column remnants appear grayed). If some text is wider than column, OllyDbg replaces last visible character in the column by symbol '>'.

To file - copies selected part of the code to file. There apply same rules as when you copy selection to clipboard, but the size of data you can copy is unlimited.

Select all - selects whole code displayed in Disassembler.

Select procedure - selects current recognized procedure.

Binary

Edit (Ctrl+E) - allows you to edit selected part of the code as ASCII, UNICODE or hexadecimal string. All three edit controls are tightly connected to each other, scroll together (first visible byte is the same in all 3 windows) and immediately display the changes you've made in any control. By pressing Ctrl+UpArrow or Ctrl+DownArrow you can quickly go to the corresponding place in different window. Maximal length of edited code is 256 bytes. Incomplete characters are displayed as red question marks. If hexadecimal string contains odd number of nibbles, OllyDbg completes it with 0. When the "Keep size" option is on, you are not allowed to insert or delete characters or write over the end of selected code.

Fill with 00's – fills selected part of code with zeros.

Fill with NOP's – fills selected part of code with NOPs

Binary copy – copies selected part of code as a hexadecimal ASCII dump to the clipboard.

Binary paste – pastes hexadecimal dump from clipboard to selection. OllyDbg scans text on clipboard and extracts hexadecimal digits (0..9, A..F, a..f), ignoring all other symbols. Code outside the selected area remains unchanged. If last byte contains single hex digit, it is ignored. For example: "part of code" is interpreted as AF CD and not as AF CD 0E.

Copy with masked fixups - same as binary copy, but substitutes all fixuped addresses with question marks. Facilitates search for similar code fragments. See also Search for binary strings.

Modify byte - allows you to edit contents of selected byte constant as a decimal signed, decimal unsigned or hexadecimal number.

Modify integer - allows you to edit contents of selected integer constant as a decimal signed, decimal unsigned or hexadecimal number.

Modify float - allows you to edit contents of selected floating-point constant.

Modify MMX - allows you to edit contents of selected MMX constant as a collection of decimal signed, decimal unsigned or hexadecimal fields.

Modify 3DNow! - allows you to edit contents of selected 3DNow! constant as a pair of floating-point or hexadecimal numbers.

Modify SSE - allows you to edit contents of selected SSE constant as a set of 4 floating-point or hexadecimal numbers.

Undo selection (Alt+BkSpc) - substitutes selected part of the code with the corresponding portion of backup data. Available only when backup data exists and differs from selected code.

Assemble (Space) - allows you to edit or overwrite existing code with one or several commands in assembler language. For details, see Assembler.

Label (:) - allows you to assign a user-defined label to the first selected address.

Edit label (:) - allows you to edit or erase the user-defined label assigned to the first selected address.

Comment (;) - allows you to add comment to the first selected address.

Edit comment (;) - allows you to edit or erase user-defined comment assigned to the first selected address.

Breakpoint

Toggle (F2) - toggles INT3 breakpoint on the first selected command.

Conditional (Shift+F2) - allows to set conditional breakpoint on the first selected command.

Conditional log (Shift+F4) - allows to set logging breakpoint. For more details, see Breakpoints.

Message breakpoint on WinProc - allows to edit active message breakpoint. Message breakpoint can be set from the Windows window.

Run to selection (F4) - sets one-shot breakpoint on the first selected command and continues execution of debugged program. If OllyDbg stops execution before the program reached this command, one-shot breakpoint still remains active. If necessary, you can remove it from Breakpoints window.

Memory, on access - sets memory breakpoint on the selected part of memory. Program stops each time the memory is accessed. OllyDbg supports single memory breakpoint of any size. Memory breakpoint can significantly slow down the execution. Under Windows 95/98, memory breakpoint may crash debugged program when system routines access memory blocks containing breakpoint. Use it as a last resort.

Memory, on write - sets memory breakpoint on the selected part of memory. Program stops each time it attempts to write to this memory. OllyDbg supports single memory breakpoint of any size. Memory breakpoint can significantly slow down the execution. Under Windows 95/98, memory breakpoint may crash debugged program when system routines access memory blocks containing breakpoint. Use it as a last resort.

Remove memory breakpoint - removes memory breakpoint.

Remove SFX memory breakpoint - stops search for real entry of self-extractable (SFX) program. This search uses memory breakpoint of a special type.

Hardware, on execution - sets hardware memory breakpoint on the first selected byte. Program stops each time it tries to execute command that begins with this byte. Hardware breakpoints are available only under Windows ME, NT or 2000. 80x86 processors support up to 4 hardware breakpoints. If OllyDbg is unable to find free slot for the hardware breakpoint, it asks you to remove some existing breakpoint. You can set hardware breakpoints on write or access in the CPU Dump pane.

Remove hardware breakpoint - removes hardware breakpoint set on the first selected byte.

#**Set real SFX entry here** - declares first selected command as a real entry point of the unpacked self-extractable program. If real SFX entry is declared and option "Use real entry from previous run" is activated, OllyDbg can quickly bypass self-extractor and stop on real entry.

Hit trace - commands that manipulate hit trace, available only for analyzed code.

Add selection - request hit trace on selected piece of code.

Add procedure - request hit trace on the current procedure.

Add all recognized procedures - request hit trace on all procedures recognized by Analyzer in the code displayed in Disassembler. To avoid crashes, I recommend that you select strict or heuristical procedure recognition.

Remove from selection - removes hit trace from selection. If there is a forced run trace, removes it too.

Remove from module - removes hit trace from the code section of the module that is currently selected in Disassembler. If there is a forced run trace, removes it too.

Mark selection as not traced - marks all selected commands with activated hit trace as not hit.

Mark module as not traced - marks all commands with activated hit trace within the code section of the module selected in Disassembler as not hit.

Run trace - commands that manipulate run trace, available only when code is analyzed.

Add selection - forces run trace on selected piece of code and simultaneously request hit trace.

Add procedure - forces run trace on the current recognized procedure and simultaneously request hit trace.

Add branches in procedure - forces run trace on all recognized jump or call destinations and removes run trace from all other commands in the current procedure. Simultaneously it requests hit trace on the whole procedure.

Add entries of all procedures - forces run trace on entry points of the recognized procedures and removes it from all other commands in the current module. Simultaneously it requests hit trace on all recognized procedures.

Skip selection when tracing - excludes selected quasi-linear piece of code from the run trace. When run trace encounters excluded code, it sets temporary breakpoint at the end of selection and runs code at once. This significantly accelerates run trace.

Set condition (Ctrl+T) - allows to set condition to pause run trace.

Profile current module - opens window with profile data of the current module.

Global profile - opens window with profile data of the whole application. Gathering global profile data may be time-consuming.

Remove from selection - removes forced run trace from selection.

Remove from module - removes forced run trace from the code section of the module that is currently selected in Disassembler.

Follow (ENTER) - follows jump, call, return or switch destination. See also Command history.

Follow immediate constant - if immediate constant in the command points to code, follows address.

Follow SE handler (ENTER) - if actual command installs structured exception handler (SEH), follows entry point of handling routine.

New origin here (Ctrl+Gray *) - sets EIP of the currently selected thread to the address of the first selected byte. You can undo this operation if you go to Registers pane and select EIP.

Go to

Origin (*) - goes to the address contained in EIP of the current thread.

Previous (-) - goes to the previous address in the command history. You can't browse run history when run trace buffer is open.

Previous run trace record (-) - goes to the previous record in the run trace buffer.

Next (+) - goes to the next address in the command history. You can't browse run history when run trace buffer is open.

Next run trace record (+) - goes to the next record in the run trace buffer.

Expression (Ctrl+G) - allows you to follow hexadecimal address or result of expression. Dialog keeps several last entered addresses. To facilitate distinguishing, you can comment addresses, simply type any text separated from address or expression by semicolon.

Previous procedure (Ctrl+Minus) - goes to the beginning of the previous recognized procedure.

Next procedure (Ctrl+Plus) - goes to the beginning of the next procedure.

Previous reference (Alt+F7) - goes to the previous found reference. Selection in References window moves synchronously with Disassembler.

Next reference (Alt+F8) - goes to the next found reference.

Source file (Ctrl+F5) - opens source window and displays code corresponding to the first selected command. Currently this option is available only if executable file contains debugging information in Borland format.

Switch base,
Default case,
Case xxxx - if command is a switch base or one of its cases, these menu items navigate between remaining elements of the recognized switch.

More cases... - if number of cases in a switch exceeds 10, opens window that displays all cases in a switch.

CALL from xxxx,
JMP from xxxx,
JNZ from xxxx,
JMP [ ] from xxxx etc. - goes to command that jumps to or calls selected command. This can be direct or indirect local (intramodular) call, direct unconditional jump, conditional jump or table switch. Notice that calls from different modules are not listed, even if they are present in call tree.

More jumps and calls... (Ctrl+J) - if number of jumps and calls to the current location exceeds 10, opens window that displays all jumps and calls to selected command. Shortcut is always active, even if this item doesn't appear in menu.

Call DLL export - invokes Call export dialog. Available only if you debug standalone DLL and first selected line is entry point of exported function in this DLL.

Thread - in multithread applications, allows quick navigation between different threads.

Follow in Dump

Selection - reopens memory block in CPU Dump and follows first selected address.

Constant - follows immediate constant in the Dump pane.

Address constant - follows constant which is part of address in the Dump pane. For example, if currently selected command is MOV [ESI+123456],543210, Dump will display contents of memory starting from address 0x123456.

Immediate constant - follows immediate constant in the Dump pane. For example, if currently selected command is MOV [ESI+123456],543210, Dump will display contents of memory starting from address 0x543210.

Implicit stack address - follows stack location implicitly addressed by ESP, like in commands PUSH and RET.

Memory address,
First address,
Second address - follows address in the Dump pane. For example, if currently selected command is MOV [ESI+123456],543210 and ESI contains 0x10, Dump will display contents of memory starting from address 0x123466. Some commands, like MOVS, have two memory operands. In this case, first address is the destination and second address is the source.

View call tree (Ctrl+K) - opens Call tree window that displays all known calls to current procedure and all procedures called from the current procedure. If you want to extend tree into different

modules, please analyze them.

Search for

Name (label) in current module (Ctrl+N) - displays table containing all names (exports, imports, library, user-defined) defined or used in the current module.

Name in all modules - displays table cointaining all known names.

#**Command** (Ctrl+F) - allows you to search for assembler command. OllyDbg tries to find all possible encodings. For example, if you search for MOV EAX,[123456], it will look for both **A1 56341290** and **8B05 56341200**. You can also specify imprecise commands, like MOV R32,[CONST] - which fits both MOV EAX,[10000] and MOV ESI,[123456]. This search, however, cannot find some more complicated address forms. For best results, analyze code before starting search.

Sequence of commands (Ctrl+S) - allows you to search for a sequence of assembler commands. This sequence may include imprecise commands and matching registers and allows to omit intermediate commands.

#**Constant** - allows you to find for a constant within the code. This constant can be part of address, immediate constant, offset for relative jump or element of switch table. For best results, analyze code before starting search.

Binary string (Ctrl+B) - displays dialog allowing to specify search pattern. Maximal size of search pattern is 256 bytes. You can exclude some bytes or nibbles from the comparison. For example, if you specify pattern **12 ?? ?6 78**, it will match both **12 34 56 78** and **12 00 06 78**, but not **12 34 55 78**. You can also ignore case of ASCII/UNICODE characters.

Modified command - searches for the next command that differs from the backup.

Trace hit - searches for the next contiguous block of commands that are marked as executed (hit) in the hit trace.

Next (Ctrl+L) - repeats last search from the selection.

All intermodular calls - searches for all commands that call (directly or indirectly, may be via several intermediate jumps) functions residing in other modules. Especially useful to find calls to API functions loaded by *GetProcAddress()*.

All commands - allows to find all commands that match specified assembler pattern.

All sequences - allows to find all sequences of commands that match specified pattern.

All constants - allows to find all instances of the specified constant in the code section of the current module.

All switches - displays table that lists all switches recognized in the current module.

#**All referenced text strings** - searches for all ASCII and UNICODE strings that are referenced in the code section of the current module or are embedded in this section.

#**User-defined label** - displays table of all user-defined labels in the current module.

#**User-defined comment** - displays table of all user-defined comments in the current

module.

#**Last record in run trace** - searches for the most recent occurrence of the first selected command in the run trace buffer.

#**Find references to** - these commands search references to the specified item in code section of the module opened in Disassembler window. For best results, analyze code before searching for references. Following search items are supported:

Selected command (Ctrl+R),
Selected address (Ctrl+R) - first selected address;

Selected block - selected range;

Immediate constant - immediate constant which is a part of the first selected command;

Address constant - address constant in the first selected command;

Call destination - call destination of the first selected command;

Jump destination - jump destination of the first selected command;

Call constant,
Jump constant - constant part of destination address in the first selected command.

Stop till return (Esc) - stops execution till return.

Stop tracing (Esc) - stops run tracing.

Stop animation (Esc) - stops animation.

View

Source file (Ctrl+F5) - opens source window and displays code corresponding to the first selected command. Currently this option is available only if executable file contains debugging information in Borland format.

Original comments - displays comments in the fourth column of Disassembler window. If bar is visible, press comment bar to toggle between comments, source and profile.

Source as comments - displays lines of source code in the fourth column of Disassembler window (normally displaying comments). If bar is visible, press comment bar to toggle between comments, source and profile. This option is available only if executable file contains debugging information in Borland's format.

Profile as comments - displays number of times each command appears in the run trace buffer. If bar is visible, press comment bar to toggle between comments, source and profile. This option is available only when profile data is open.

Executable file - displays dump of the executable file at offset that corresponds to the first selected command. If selection is not in the file, dump is positioned at offset 0.

Absolute address - displays absolute addresses in the first column.

Relative address - displays addresses relative to the currently selected. Alternatively,

doubleclick base address in the first column.

Module 'xxx' - displays executable code of selected module.

Copy to executable

Selection - copies selection to the executable file. OllyDbg adjusts fixups and warns you if this operation may cause errors.

All modifications - copies all highlighted modification (i.e. differences between actual code and global backup) to executable file.

Analysis

Analyze code (Ctrl+A) - analyzes code section of the module opened in Disassembler window. Other parts of OllyDbg work more reliably if analysis data is available.

Remove analysis - discards analysis data for current module.

Scan object files (Ctrl+O) - allows you to select object files or libraries and locate their positions in the code section of the module opened in Disassembler window.

Remove object scan - discards results of object scan.

#**Assume arguments** - allows to treat first selected command as entry point of a function with predefined arguments. Currently available function types are:

WinProc(hWnd,msg,wParam,lParam) - windows function that processes messages
WinMain(hInst,hPrevInst,CmdLine,ShowState) - program entry point
DllEntryPoint(hInst,CallReason,pReserved) - DLL entry point
Format(format,...) - function similar to printf
Sformat(ptr,format,...) - function similar to sprintf
StdFunc0(void) - function without arguments
StdFunc1(int) - function with single argument
.....
StdFunc8(int,int,int,int,int,int,int,int) - function with 8 arguments

Remove analysis from selection (BkSpc) - removes analysis from selected block. Useful if Analyzer has misinterpreted code as data.

During next analysis, treat selection as - sets decoding hints.

Help on symbolic name - if first selected line contains symbolic name and API help file is attached to OllyDbg, attempts to open help topic on the symbolic name.

Appearance - see detailed description here.

$#K Appearance menu

This menu is available in most OllyDbg windows. It allows to customize appearance of separate windows or panes.

Always on top - keeps window on the top of all other MDI windows. Only one MDI window at a time may be declared as topmost. Notice that this option works incorrectly with MDI windows created by old plugins.

Show bar - displays bar with column titles over the window. In some windows, bar allows to sort the contents of the table, or change the presentation of the data, or reswitch between actual data and backup.

Hide bar - removes bar from the window.

Show horizontal scroll - displays horizontal scroll that allows to scroll window to the left or to the right. Alternatively, one can use left and right arrows.

Hide horizontal scroll - removes horizontal scroll.

Default columns - resets width of all columns in the table to default.

Font,
Font (this),
Font (all) - allows to change the font in the window, in the currently selected pane or in all panes of CPU window. By default, OllyDbg defines following fixed-pitch fonts:

OEM fixed font	- standard font
Terminal	- compact well-readable font
System fixed font	- standard font
Courier	- supports UNICODE
Lucida	- supports UNICODE, usually not available in Windows 95/98

You can add your own fonts and modify existing definitions.

Colors,
Colors (this),
Colors (all) - allows to select color scheme in the window, in the currently selected pane or in all panes of CPU window. By default, OllyDbg supports following color schemes:

Black on white	- black letters on white background
Yellow on blue	- yellow letters on dark-blue background
Marine	- different tones of blue
Mostly black	- white letters on black background

You can create your own schemes and edit existing.

Highlighting - allows to select or disable highlighting for disassembled code. You can create your own highlighting schemes and edit existing. Two predefined schemes, **Chrismas tree** and **Jumps'n'calls**, are best viewed with **Black on white** color scheme.

$#K# Dump

Dump window displays contents of memory or file. You can select one of several predefined formats: byte, text, integer, float, address, disassembly or PE Header.

All dump windows support backup, search and edit functions. Dump pane of CPU window allows you to define labels, set memory breakpoints, find references to data in executable code and open image of selected memory in the executable file (.exe or .dll). Dump menu displays only relevant subset of available commands.

If backup is available, press Address/Backup in the bar to toggle display mode between backup and normal view. Other bar buttons allow you to change dump modes.

Like Disassembler, Dump keeps long history of visited memory locations. You can navigate through the history by pressing buttons '**+**' and '**-**'.

To scroll data bytewise, hold **Ctrl** key and press **Up**/**Down** arrows.

$#K Dump menu

Backup - see here

Undo selection (Alt+Backspace) - substitutes selected part of the data with the corresponding portion of backup. Available only when backup exists.

Copy

> **To clipboard** (Ctrl+C) - copies selection to clipboard in the currently selected format. To exclude some column from the copy, reduce its width to the minimal possible (in this case column remnants appear grayed). If some text is wider than column, OllyDbg replaces last visible character in the column by '>'.
>
> **To file** - copies selection directly to file. There apply same rules as when you copy selection to clipboard, but the size of data you can copy is unlimited.
>
> **Select all** - selects whole memory block.

Binary

> **Edit** (Ctrl+E or hexadecimal digit) - allows you to edit selected data as ASCII, UNICODE or hexadecimal string. All three edit controls are tightly connected to each other, scroll together (first visible byte is the same in all 3 windows) and immediately display the changes you've made in any control. By pressing Ctrl+UpArrow or Ctrl+DownArrow you can quickly go to the corresponding place in different window. Maximal length of edited data is 256 bytes. Incomplete characters are displayed as red question marks. If hexadecimal string contains odd number of nibbles, OllyDbg completes it with 0. When the "Keep size" option is on, you are not allowed to insert or delete characters or write over the end of selected code.
>
> **Fill with 00's** – fills selected part of data with zeros.
>
> **Fill with FF's** – fills selected part of data with 0xFF.
>
> **Binary copy** – copies selected part of data as a hexadecimal ASCII dump to the clipboard.
>
> **Binary paste** – pastes hexadecimal dump from clipboard to selection. OllyDbg scans text on clipboard and extracts hexadecimal digits (0..9, A..F, a..f), ignoring all other symbols. Data outside the selected area remains unchanged. If last byte contains single hex digit, it is ignored. For example: "part of data" is interpreted as AF DA and not as AF DA 0A.
>
> **Copy with masked fixups** - same as binary copy, but substitutes all fixuped addresses with question marks. Facilitates search for similar code fragments. See also Search for binary strings.

Modify - allows you to modify the value of the first selected item (integer or float).

Assemble - allows you to overwrite the data, starting from the first selected byte, with one or several commands in assembler language. For details, see Assembler.

Modify byte - allows you to edit contents of selected byte as a decimal signed, decimal unsigned or hexadecimal number.

Modify integer - allows you to edit contents of selected integer as a decimal signed, decimal unsigned or hexadecimal number.

Modify float - allows you to edit contents of selected floating-point number.

Label (:) - allows you to assign a user-defined label to the first selected address.

Edit label (:) - allows you to edit or erase the user-defined label assigned to the first selected address.

Breakpoint

Memory, on access - sets memory breakpoint on the selected part of data. Program stops each time the memory is accessed. OllyDbg supports single memory breakpoint of any size. Memory breakpoint can significantly slow down the execution. Under Windows 95/98, memory breakpoint may crash debugged program when system routines access memory blocks containing breakpoint. Use it as a last resort.

Memory, on write - sets memory breakpoint on the selected part of data. Program stops each time it attempts to write to this memory. OllyDbg supports single memory breakpoint of any size. Memory breakpoint can significantly slow down the execution. Under Windows 95/98, memory breakpoint may crash debugged program when system routines access memory blocks containing breakpoint. Use it as a last resort.

Remove memory breakpoint - removes memory breakpoint.

Remove SFX memory breakpoint - stops search for real entry of self-extractable (SFX) program. This search uses memory breakpoint of special type.

Hardware, on access - sets hardware memory breakpoint starting from the first selected byte. Breakpoint may cover 1, 2 or 4 bytes and must be aligned. Program stops each time it accesses the covered memory. Hardware breakpoints are available only under Windows ME, NT or 2000. 80x86 processors support up to 4 hardware breakpoints. If OllyDbg is unable to find free slot for the hardware breakpoint, it asks you to remove some existing breakpoint.

Hardware, on write - sets hardware memory breakpoint starting from the first selected byte. Breakpoint may cover 1, 2 or 4 bytes and must be aligned. Program stops each time it tries to write to the covered memory. Hardware breakpoints are available only under Windows ME, NT or 2000. 80x86 processors support up to 4 hardware breakpoints. If OllyDbg is unable to find free slot for the hardware breakpoint, it asks you to remove some existing breakpoint.

Hardware, on execution - sets hardware memory breakpoint on the first selected byte. Program stops each time it attempts to execute command that begins with this byte. Hardware breakpoints are available only under ME, NT or 2000. 80x86 processors support up to 4 hardware breakpoints. If OllyDbg is unable to find free slot for the hardware breakpoint, it asks you to remove some existing breakpoint.

Search for

Command (Ctrl+S) - allows you to search for assembler command. OllyDbg tries to find all possible encodings. For example, if you search for MOV EAX,[123456], it will look both for sequence **A1 56341290** and **8B05 56341200**. You can also specify imprecise commands, like MOV R32,[CONST] - which fits both MOV EAX,[10000] and MOV ESI,[123456]. This search, however, cannot find some more complicated address forms.

Integer,
Address - allows you to search for a 16- or 32-bit constant within the data. Search can be aligned, when OllyDbg goes through data as displayed in the window (i.e. with step equal to

the size of the displayed element), and unaligned, when it tries every byte.

Float - allows you to search for a 32-, 64- or 80-bit float constant within the data. Search can be aligned, when OllyDbg goes through data as displayed in the window (i.e. with step equal to the size of the displayed element), and unaligned, when it tries every byte. You can also allow for 0.1% error margin. This is especially helpful when you search for a non-integer constant, like 2/3=0.6666666...

Binary string (Ctrl+B) - displays dialog allowing to specify search pattern. Maximal size of the pattern is 256 bytes. You can exclude some bytes or tetrades from the comparison. For example, if you specify pattern **12 ?? ?6 78**, it will match both **12 34 56 78** and **12 00 06 78**, but not **12 34 55 78**. You can also ignore case of ASCII characters.

Modified data - searches for next item that differs from the backup.

Next (Ctrl+L) - repeats last search from the selection.

Name (label) (Ctrl+N) - displays table containing all labels (exports, imports, library, user-defined) in the current module.

Follow DWORD in Disassembler,
Follow in Disassembler - follows contents of the first selected doubleword in the Disassembler pane.

Follow DWORD in Dump,
Follow in Dump - follows contents of the first selected doubleword in the CPU Dump pane.

#**Find references** (Ctrl+R) - searches for all references to the selected address range in the code section of the module that contains data opened in Dump pane. For best results, analyze code before searching for references.

View executable file - displays dump of the executable file at offset that corresponds to the first selected byte. If selection is not in the file, dump is positioned at offset 0.

Copy to executable file - copies selection to the executable file. OllyDbg adjusts fixups and warns you if this operation may cause errors. Be careful if you copy parts of data section because they may contain important initializations already changed by code executed so far.

Save file - if this is a dump of the file, writes file to disk.

Go to expression,
Go to address,
Go to offset - allows to position dump at specified address or value of expression (or file offset if this is a dump of the file).

Go to

Expression - position dump at value of specified expression.

Previous (Minus) - go to previous selection in Dump history.

Next (Plus) - go to next selection in Dump history.

View image in Disassembler,
View image in CPU Dump - if Dump displays executable (.exe or .dll) file of some loaded module, shows its image in memory.

#Hex

Hex/ASCII (16 bytes) - displays data as hexadecimal and ASCII dump, 16 bytes per line.

Hex/ASCII (8 bytes) - displays data as hexadecimal and ASCII dump, 8 bytes per line.

Hex/UNICODE (16 bytes) - displays data as hexadecimal and UNICODE dump, 16 bytes per line.

Hex/UNICODE (8 bytes) - displays data as hexadecimal and UNICODE dump, 8 bytes per line.

Text

ASCII (64 chars) - displays data as ASCII dump, 64 characters per line.

ASCII (32 chars) - displays data as ASCII dump, 32 characters per line.

UNICODE (64 chars) - displays data as UNICODE dump, 64 characters (128 bytes) per line. To adjust alignment of data in 1-byte steps, press Ctrl+UpArrow or Ctrl+DnArrow.

UNICODE (32 chars) - displays data as UNICODE dump, 32 characters (64 bytes) per line.

Short

Signed decimal - displays data as 16-bit signed decimal numbers (8 per line). To adjust alignment of data in 1-byte steps, press Ctrl+UpArrow or Ctrl+DnArrow.

Unsigned decimal - displays data as 16-bit unsigned decimal numbers (8 per line).

Hex - displays data as 16-bit hexadecimal numbers (8 per line).

Long

Signed decimal - displays data as 32-bit signed decimal numbers (4 per line). To adjust alignment of data in 1-byte steps, press Ctrl+UpArrow or Ctrl+DnArrow.

Unsigned decimal - displays data as 32-bit unsigned decimal numbers (4 per line).

Hex - displays data as 32-bit hexadecimal numbers (4 per line).

Address - displays data as 32-bit addresses and tries to decode the meaning of the address. Stack window also uses this form.

Address with ASCII dump - displays data both as 32-bit address and ASCII dump and tries to decode the meaning of the address.

Address with UNICODE dump - displays data both as 32-bit address and UNICODE dump and tries to decode the meaning of the address.

Float

32-bit float - displays data as 32-bit floating-point numbers (4 in each line). If data is not a valid float, OllyDbg displays type of data (NAN, INF) followed by hexadecimal dump. To adjust alignment of data in 1-byte steps, press Ctrl+UpArrow or Ctrl+DnArrow.

64-bit double - displays data as 64-bit floating-point numbers (2 in each line). If data is not a valid float, OllyDbg displays type of data (NAN, INF) followed by hexadecimal dump.

80-bit long double - displays data as 64-bit floating-point numbers (2 in each line). If data is not a valid float, OllyDbg displays type of data (NAN, INF, UNORM) followed by hexadecimal dump.

Disassemble - displays data as a code.

Special

PE header - displays data as a commented COFF header of the executable file.

Appearance - see detailed description here

$#K# Stack window

Stack pane of CPU window displays stack of actual thread. Normally, when debugged program pauses execution, stack window scrolls so that element actually pointed by ESP gets topmost. This address has black background. In some cases it may be more convenient to disable (lock) scrolling. You can do it either from the pop-up menu or by pressing button "Address" in the bar.

OllyDbg can recognize *possible* return addresses. If address in stack points to a location immediately following CALL, OllyDbg highlights this line. If possible, it also reports address of the function the CALL operator is pointing to. However, you must be very careful. Frequently (not to say: *mostly*) such address is simply a remnant of some previous call.

Stack pane traces chain of structured exception handlers (SEH) and comments addresses of handlers and pointers to next element in this chain. Start address of SEH chain is the first doubleword of the data block of the corresponding thread. To locate it, go to expression [FS:0].

You can ask OllyDbg to trace stack frames. Procedures create stack frame when they execute PUSH EBP; MOV EBP,ESP or ENTER xxx,0. Again, same warning: I do not guarantee that found stack frames are actual.

If EIP points to call or jump to the known function or to the known function itself, OllyDbg will decode arguments passed to this function in the stack.

If debugging information is available, OllyDbg will mark arguments and local variables of the function on the stack. They are available only after new stack frame was created.

Stack menu contains following entries:

Address

> **Absolute** - tells OllyDbg to display absolute hexadecimal addresses.
>
> **Relative to selection** - tells OllyDbg to display stack addresses relative to the actually selected address. If you change selection afterwards, base address remains unchanged. Alternatively, doubleclick address.
>
> **Relative to ESP** - tells OllyDbg to display stack addresses relative to the contents of ESP in actual thread. If contents of ESP changes, addresses follow modified value.
>
> **Relative to EBP** - tells OllyDbg to display stack addresses relative to the contents of EBP in actual thread. If contents of EBP changes, addresses follow modified value.

Show ASCII dump - displays additional column with ASCII dump.

Show UNICODE dump - displays additional column with UNICODE dump.

Hide dump - removes column with ASCII or UNICODE dump.

Lock stack - disables automatical stack scrolling to the actual contents of ESP each time the execution is paused. Alternatively, press button "Address" in the bar. When scrolling is disabled, button changes to "Locked".

Unlock stack - enables automatical stack scrolling to the actual contents of ESP each time the execution is paused. Alternatively, press button "Locked" in the bar. When scrolling is enabled, button changes to "Address".

Copy to clipboard (Ctrl+C) - copies selection to clipboard. To exclude some column from the copy, reduce its width to the minimal possible (column remnants appear grayed). If some text is wider than column, OllyDbg replaces last visible character in the column by '>'.

Modify - allows to edit the contents of first selected doubleword.

Edit (Ctrl+E) - allows to edit the contents of selection in ASCII, UNICODE or hexadecimal format.

Push DWORD - decreases ESP by 4.

Pop DWORD - increases ESP by 4.

Search for address - allows you to search for a constant on the stack. Search can be aligned or unaligned. Usually all constants pushed onto stack are aligned.

Search for binary string (Ctrl+B) - displays dialog allowing to specify search pattern. Maximal size of search pattern is 256 bytes. You can exclude some bytes or nibbles from the comparison. For example, if you specify pattern **12 ?? ?6 78**, it will match both **12 34 56 78** and **12 00 06 78**, but not **12 34 55 78**. You can also ignore case of ASCII/UNICODE characters.

Go to ESP (Asterisk) - follows contents of ESP register of actual thread in Stack window.

Go to EBP - follows contents of EBP register of actual thread in Stack window.

Go to expression (Ctrl+G) - position stack at value of specified expression.

Follow in Disassembler (Enter) - follows contents of first selected doubleword in Disassembler pane.

Follow in Dump - follows contents of first selected doubleword in Dump pane.

Follow in Stack - follows contents of first selected doubleword in Stack pane.

Appearance - see detailed description here

$#KExecutable modules window

Executable modules window (keyboard shortcut: **Alt+E**) lists all executable modules currently loaded by debugged process. It also displays useful information, like module size, entry address, module version or path to executable file. Some information, like decimal module size, symbolical name of entry point or whather module is system is normally hidden. To see it, increase width of corresponding columns. Pop-up menu supports following options:

Actualize - re-scans modules and removes highlighting from the new modules. In most cases, OllyDbg takes care by himself.

View memory - opens Memory window and scrolls it to the first memory block belonging to the module's image.

View code in CPU (Enter) - opens module's executable code in Disassembler.

Follow entry - follows entry point of the module in Disassembler.

Dump data in CPU - opens module's data section in CPU Dump.

View names (Ctrl+N) - displays table containing all names (exports, imports, library, user-defined) defined or used in the current module.

Mark as system DLL,
Mark as non-system DLL - marks selected module as system or non-system. Selecting module as systems allows you to exclude it from Run trace, significantly accelerating it. By default, system modules are modules that reside in system directory (usually *c:\windows\system* on Windows 95/98, *c:\winnt\system32* on NT/2000/XP).

Update .udd file now - writes all module-dependent data to file <modulename>.udd. .udd files keep breakpoints, labels, comments, watches, analysis and so on between sessions. OllyDbg automatically creates .udd file when module unloads.

View executable file - displays dump of the executable file.

View all resources - displays list of all resources defined in the module together with brief information. OllyDbg doesn't support resources as a separate entity; all you can is to dump and binary edit them.

View resource strings - displays list of resource strings and their identifiers.

View run trace profile - calculates profile for this module. See also Run trace.

Analyze all modules - allows to analyze all modules at once. Analysis extracts loads of useful information from the code; debugging is usually faster and much reliable when code is analyzed.

By doubleclicking the line, you follow executable code of the module in Disassembler.

$#K# Memory map window

Memory map window displays all memory blocks allocated by debugged program. There are no standard means to accomplish this task, so it may happen that OllyDbg unites several portions of allocated memory in one larger memory block. In most cases, however, precise resolution is not necessary. To get list of memory chunks requested by application via calls to *GlobalAlloc()*, *LocalAlloc()* etc., use Heap list.

If memory block is a section of some executable module, OllyDbg reports which kinds of data this block contains: code, data, resources etc.

There are some differences between Windows 95/98 and Windows NT/2000. Under Windows 95/98, OllyDbg is unable to display names of mapped files. Additionally, Windows 95/98 limits allowed types of memory access to read and write, whereas Windows NT/2000 has much broader palette of possibilities, including execute access, copy-on-write and guarding flag. OllyDbg ignores copy-on-write attribute.

If OllyDbg recognizes that program has allocated new or reallocated existing memory block, it highlights corresponding record in memory map window. To reset highlighting, select *Actualize* from pop-up menu.

You can invoke Memory window by pressing **Alt+M**.

Following pop-up menu items are available:

Actualize - updates list of allocated memory and removes highlighting from new memory blocks.

View in Disassembler - opens memory block in Disassembler. This option is available only when memory block contains executable code or self-extractor of some module.

Dump in CPU - dumps contents of memory block in the CPU Dump pane.

Dump - dumps contents of memory block in a separate window. If type of memory block is known, OllyDbg automatically selects dump format.

View all resources - if block contains resource data, lists all resources and related data. OllyDbg doesn't support resources as a separate entity; all you can is to dump and binary edit them.

View resource strings - if block contains resource data, lists all resource strings with their identifiers.

Search - allows you to search all memory blocks, starting from selection, for the occurrence of binary string. If string is found, OllyDbg dumps found memory block. Memory map and dump windows share the same search pattern, so you can immediately continue search for the next occurrence in the appearing dump. You can close this dump window by pressing Esc.

Search next (Ctrl+L) - repeats last search.

Set break-on-access (F2, available only under Windows NT/2000) - protects the whole memory block. After break happens, OllyDbg stops debugged program and removes breakpoint. This breakpoint is especially useful if you want to catch call or return to some module.

Remove break-on-access (F2) - removes break-on-access protection from the memory block.

Set memory breakpoint on access - sets memory breakpoint on the whole memory block.

Program will stop each time memory block is accessed. OllyDbg supports only one memory breakpoint. Under Windows 95/98, debugged program may crash when system routines access memory blocks containing memory breakpoint. Use it as a last resort.

Set memory breakpoint on write - sets memory breakpoint on the whole memory block. Program will stop each time when it writes to the memory block. Under Windows 95/98, debugged program may crash when system routines access memory blocks containing memory breakpoint. Use it as a last resort.

Remove memory breakpoint - removes memory breakpoint.

Remove SFX memory breakpoint - stops search for real entry of self-extractable (SFX) program. This search uses memory breakpoint of special type.

Set access - sets desired memory protection attributes to the whole memory block. Possible options are:

No access
Read only
Read/write
Execute
Execute/read
Full access

Copy to clipboard

Whole line - copies selected record to clipboard as a multiline text with explanations. To exclude some column from the copy, reduce its width to minimum (the remaining part of column is grayed).

Whole table - copies the whole memory map to clipboard as a multiline text. First line of this text contains window's title ("Memory map"), second line - titles of the columns, all subsequent lines - records with memory data. Copy preserves width of columns. To exclude some column from the copy, reduce its width to minimum (the remaining part of column is grayed).

Appearance - see here.

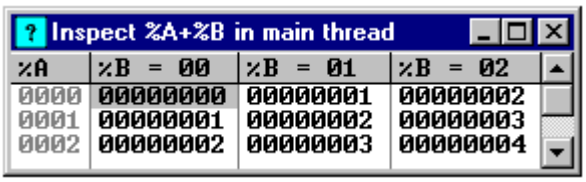
$#K# Watches and inspectors

Watch window contains several expressions. It displays their values in the second column. OllyDbg saves expressions to the .udd file of main module, so they are available in the next debugging session.

An **inspector** is a stand-alone window that can show the contents of some variable, 1- or 2-dimensional array or even selected items of array of structures. The expression is basically the same as for watch but can include two parameters: %A and %B. When you define inspector, you can specify limits for these parameters. OllyDbg then substitues all possible combinations of %A and %B into expression, starting from 0 and up to the limit (not included), and displays results in the table. Limit for %B (number of columns) cannot exceed 16.

For example, if you specify expression %A+%B and limit %A and %B to 3, you will get the following table:

{bmc bm3.WMF}



$#K Threads

OllyDbg features simple yet effective thread management. If you step, trace, execute till return or execute till selection, it stops all threads except for the actual, and resumes actual thread even if it was suspended. In this case, if you suspend or resume threads manually, actions are postponed. If you run debugged application, OllyDbg restores original thread states. (Notice that from the Debugger's point of view, hit trace is equivalent to the free run).

In accordance with this scheme, Threads window distinguishes following 5 possible thread states:

Active	- thread is running or paused on debugging event
Suspended	- thread is suspended
Traced	- thread is suspended, but OllyDbg steps through it
Paused	- thread is active, but OllyDbg temporarily suspended it to step through other thread
Finished	- process is terminated

Threads window also displays last thread error (as returned by call to *GetlastError*) and time this thread spent in user and in system mode (NT/2000/XP only). It highlights the identifier of main thread.

Following pop-up menu items are available:

Actualize - marks all threads as old.

Suspend - suspends thread.

Resume - resumes previously suspended thread.

Set priority - adjust priority of the thread within the process. Following options are available:

Idle	- lowest possible priority of the thread within the process
Lowest	
Low	
Normal	
High	
Highest	
Time critical	- highest possible priority

Open in CPU (doubleclick) - displays current state of selected thread in CPU window.

Copy to clipboard

Whole line - copies selected record to clipboard as a multiline text with explanations. To exclude some column from the copy, reduce its width to minimum (the remaining part of column is grayed).

Whole table - copies the whole memory map to clipboard as a multiline text. First line of this

text contains window's title ("Memory map"), second line - titles of the columns, all subsequent lines - records with memory data. Copy preserves width of columns. To exclude some column from the copy, reduce its width to minimum (the remaining part of column is grayed).

Appearance - see here.

$#K Evaluation of expressions

OllyDbg supports very complex expressions. Formal grammar of expressions is described at the end of this topic, but honestly - you are not interested in it, are you? So I'll begin with examples:

10 - constant 0x10 (unsigned). All integer constants are assumed hexadecimal unless followed by a decimal point;

10. - decimal constant 10 (signed);

'A' - character constant 0x41;

EAX - contents of register EAX, interpreted as unsigned number;

EAX. - contents of register EAX, interpreted as signed number;

[123456] - contents of unsigned doubleword at address 123456. By default, OllyDbg assumes doubleword operands;

DWORD PTR [123456] - same as above. Keyword PTR is optional;

[SIGNED BYTE 123456] - contents of signed byte at address 123456. OllyDbg allows both MASM- and IDEAL-like memory expressions;

STRING [123456] - ASCII zero-terminated string that begins at address 123456. Square brackets are necessary because you display the contents of memory;

[[123456]] - doubleword at address that is stored in doubleword at address 123456;

2+3*4 - evaluates to 14. OllyDbg assigns standard C priorities to arithmetical operations;

(2+3)*4 - evaluates to 20. Use parentheses to change the order of operations;

EAX.<0. - 0 if EAX is in range 0..0x7FFFFFFF and 1 otherwise. Notice that constant 0 is also signed. When comparing signed with unsigned, OllyDbg always converts signed operand to unsigned.

EAX<0 - always 0 (false), because unsigned numbers are always positive.

MSG==111 - true if message is WM_COMMAND. 0x0111 is the code for WM_COMMAND. Use of MSG makes sense only within conditional or conditional logging breakpoint set on call to or entry of known function that processes messages.

[STRING 123456]=="Brown fox" - true if memory starting from address 0x00123456 contains ASCII string "Brown fox", "BROWN FOX JUMPS", "brown fox???" or similar. The comparison is case-insensitive and limited in length to the length of text constant.

EAX=="Brown fox" - same as above, EAX is treated as a pointer.

UNICODE [EAX]=="Brown fox" - OllyDbg treats EAX as a pointer to UNICODE string, converts it to ASCII and compares with text constant.

[ESP+8]==WM_PAINT - in expressions, you can use hundreds of symbolic constants from Windows API.

([BYTE ESI+DWORD DS:[450000+15*(EAX-1)]] & 0F0)!=0 - absolutly valid expression.

And now the formal grammar. Eeach element in braces ({}) may occur only once, order of embraced elements is not important:

expression = **memterm** | **memterm** <binary operation> **memterm**

memterm = **term** | { **sigmod sizemod prefix [** } **expression]**

term = **(expression)** | **unaryoperation memterm** | **signedregister** | **register** | **fpuregister** | **segmentregister** | **integerconst** | **floatingconst** | **stringconst** | **parameter** | **pseudovariable**

unaryoperation = ! | **~** | **+** | **-**

signedregister = register .

register = AL | BL | CL ... | AX | BX | CX ... | EAX | EBX | ECX ...

fpuregister = ST | ST0 | ST1 ...

segmentregister = CS | DS | ES | SS | FS | GS

integerconst = <decimal constant>**.** | <hexadecimal constant> | <character constant> | <symbolic API constant>

floatingconst = <floating constant>

stringconst = **"**<string constant>**"**

sigmod = SIGNED | UNSIGNED

sizemod = BYTE | CHAR | WORD | SHORT | DWORD | LONG | QWORD | FLOAT | DOUBLE | FLOAT10 | STRING | UNICODE

prefix = **term:**

parameter = %A | %B // Allowed in inspectors only

pseudovariable = MSG // Code of window message

This grammar is not too strict, there is an intrinsic ambiguity in the interpretation of [WORD [EAX]] or similar expressions. Is this a DWORD on address which is stored in two bytes on address EAX, or is this a WORD on address to be taken from 4-byte memory addressed by EAX? OllyDbg tries to add modifiers to the outermost address as long as it's possible. In our case, [WORD [EAX]] is equivalent to WORD [[EAX]].

By default, BYTE, WORD and DWORD are unsigned whereas CHAR, SHORT and LONG are signed. All general-purpose registers are unsigned. One may use explicit modifiers SIGNED and UNSIGNED (even with registers). In binary operations, if one of operands is float, another will be converted to float, else if one is unsigned, another will be also converted to unsigned. Floating-point types do not accept UNSIGNED. MASM-compatible keyword PTR after size modifier (like in BYTE PTR) is also allowed but not required. Register names and size modifiers are not case-sensitive.

You can use following C-like arithmetical operations (priority 0 is highest):

Priority	Type	Operations
0	Unary	! ~ + -
1	Multiplication	* / %
2	Addition	+ -
3	Shifts	<< >>
4	Comparisons	< <= > >=
5	Comparisons	== !=
6	Boolean AND	&
7	Boolean XOR	^
8	Boolean OR	|
9	Logical AND	&&
10	Logical OR	||

In calculations, intermediate results are kept as either DWORD or FLOAT10. Some combinations of term types and operations are not allowed. For example, QWORDs can be only displayed; STRING and UNICODE allow only + and - (as if they were C pointers) and comparison for equal/not equal with STRING, UNICODE or string constant; you cannot shift FLOAT etc.

$#K# Options

Appearance options

- General
- Defaults
- Dialogs
- Directories
- Fonts
- Colours
- Code highlighting

Debugging options (Alt+O)

- Security
- Debug
- Events
- Exceptions
- Trace
- SFX
- Strings
- Addresses
- Commands
- Disasm
- CPU
- Registers
- Stack
- Analysis 1
- Analysis 2
- Analysis 3

Just-in-time debugging

Add to Explorer

$#KGeneral options

All options

Scroll workarea (startup option!)
When this option is on and some child within the client area of the main OllyDbg window is clipped, scrollbars appear which allow you to scroll the client area, effectively increasing its size. You must restart OllyDbg for this option to take effect.

Snow-free (slower) drawing
When this option is off, OllyDbg draws directly to screen. This drawing mode is fast but causes temporary snow-like effects that may appear awkward to your eyes. When option is on, OllyDbg prepares updated image in the memory and then copies it to the screen. On some old systems, this way may be significantly slower.

Append arguments to file names in history
The File menu contains the last six debugged programs. When this option is on, OllyDbg appends arguments to the program's name, so they appear like a complete command line.

Restore window positions and appearance
When window that allows only one instance (like Memory or CPU) closes, OllyDbg saves its position, width of columns, font and color scheme to the initializations file. If this option is on, OllyDbg restores this window in the same coordinates and with same appearance.

Restore width of columns
If option is on, OllyDbg restores width of columns for windows that allow for only one instance.

Highlight sorted column
Asks OllyDbg to highlight bar button that is responsible for current sorting mode.

Show toolbar
Shows or hides toolbar. In the current version, toolbar is not customizeable.

Show status in toolbar
When this option is deselected, OllyDbg displays status of debugged application (Paused / Running / Terminated etc.) in the bottom right corner of the main window. Sometimes you may want to see as much of the application's windows as possible and move OllyDbg so that it's partially invisible. To make status visible in this case, show toolbar and check this option. Status will be displayed as a first toolbar item.

Startup size of log buffer
This startup option specifies amount of memory that OllyDbg allocates for data displayed in Log window.

$#K# Defaults

All options

Horizontal scroll bar in tables

If this option is selected, all newly created windows will contain horizontal scroll bar. Scroll bar, however, covers significant part of the precious space, reducing amount of data you can see on the screen. Most of OllyDbg windows use left and right arrows to scroll contents to left or right.

Default font

Here you select default font that will be used by all newly created windows. By default, OllyDbg creates following 5 fixed-pitch fonts:

- OEM fixed font
- Terminal 6
- System fixed font
- Courier (UNICODE)
- Lucida (UNICODE) (not available under Windows 95)

The last two support extended UNICODE character set. Notice that this setting doesn't change font in the already existing windows, or in windows which position and appearance are restored from the .ini file. You can change font in existing window by selecting Appearance|Font in pop-up menu. You can customize fonts in font options.

Default color scheme

Select color scheme that will be used in all newly created windows. OllyDbg supports 4 default color schemes:

- Black on white
- Yellow on blue
- Marine
- Mostly black

Notice that this setting doesn't change colors in the already existing windows, or in windows which position and appearance are restored from the .ini file. You can change colors in existing window by selecting Appearance|Colors in pop-up menu. You can customize color schemes in color options.

Default syntax highlighting

Select highlighting scheme that will be used as default in CPU, Run trace and all newly created dump windows, or disable syntax highlighting. OllyDbg supports up to 4 highlighting schemes. Notice that selected color scheme has no influence on highlighting colors, so some combinations of color/highlighting may be unreadable. In this case you may want to edit highlighting scheme.

$#K# Dialog options

All options

Align dialogs on selection
When option is activated, OllyDbg aligns dialogs so that they don't obscure the selection.

Font used in dialogs:

System default
If this option is on, edit controls and list boxes in dialogs use system default font.

Same as in parent window
Edit controls and list boxes in dialogs use font of the window that called this dialog.

Specified here
Here you may select font to be used in dialogs. Your choice is limited to one of the 8 customizable fixed-pitch fonts.

$#KDirectory options

All options

UDD path

Here you can specify directory where OllyDbg will save .udd files with module-related information.

> **Backup old .udd files**
> Selects whether OllyDbg should create backup copy of old .udd files.

Plugin path

Directory where OllyDbg looks for plugins.

$#K# Font options

All options

Here you can customize fonts that OllyDbg uses to draw contents of windows. First select one of 8 available fonts. Press Rename to change its internal name (as in Appearance menu). Press Change to select different fixed-pitch font, style and size.

For some fonts, reported and visible heights are too different. In this case, you can adjust the height of rows in OllyDbg windows by adding or subtracting up to 5 pixels to the reported height.

To restore default settings for selected font, press Restore defaults. If you press Cancel, OllyDbg will discard all modifications you've made.

$#K# Colour options

All options
Here you can customize colour schemes that OllyDbg uses to display contents of windows. First select one of 8 available colour schemes. If you want to change its internal name (as in Appearance menu), press rename.

Each colour scheme consists of 8 elements:

Element	Used to draw	Default in Black on white
Plain text	Ordinary text, scope brakes, current EIP	Black
Highlighted text	Highlighted text, jump path, breakpoints	Light red
Grayed text	Grayed text	Gray
Normal background	Ordinary background	White
Selected background	Selection	Light gray
Separating line	Lines between columns	Gray
Auxiliary object	Lines in hex dump, minimized columns	Light gray
Conditional breakpoint	Conditional breakpoints	Magenta

To change colour, first select element, then click on the colour you want to assign. Repeat this with all 8 elements. OllyDbg automatically saves new settings to file *ollydbg.ini*.

To restore default settings for selected colour scheme, press Restore defaults. If you press Cancel, OllyDbg will discard all modifications you've made.

$#K Code highlighting

All options

To improve the readability of the code, you may highlight different types of commands and operands in disassembled code. In the Code highlighting pane of Appearance options you can create own or edit existing highlighting schemes.

Highlighting recognizes following classes of commands:

FPU/MMX/SSE	All commands that work with floating-point, MMX or SSE operands
Cond jumps	All conditional jumps
Jumps	All unconditional jumps, including indirect
PUSH/POP	PUSH and POP commands
Calls	CALL commands
Returns	RET and RET nn commands (both near and far)
Privileged	Privileged commands that cause exception on ring 3
Bad commands	Invalid commands
Filling	Commands that are used to fill gaps between aligned procedures
Modified code	Code that differs from backup

If checkbox "Highlight operands" is chekced, operand are highlighted according to this table:

General registers	General-purpose integer registers (EAX, EBX...)
FPU/SSE registers	Floating-point, MMX, 3DNow! and SSE registers
Seg/sys registers	Segment (DS, SS...), debug (DR0...) and control (CR0...) registers
Stack memory	Memory addressed via ESP or EBP
Other memory	Any other memory operands
Address constants	Constants that address existing memory
Other constants	Any other constants

To restore default settings for selected highlighting scheme, press Restore defaults. If you press Cancel, OllyDbg will discard all modifications you've made.

You can set highlighting in the Appearance menu of Disassembler, CPU Dump or any standalone Dump window.

Notice that default highlighting schemes, Christmas tree and Jumps'n'calls, are optimized for Black on white color scheme.

$#KSecurity options

All options

When loading module-related information:

Ignore path and extention

OllyDbg keeps user-defined data, like labels, comments or breakpoints, in .udd files. Usually this file has name of executable module and resides in OllyDbg directory. However, it is perfectly legal for executable file and some DLL to have same name, or have several identically named DLLs in different directories. To avoid conflicts, OllyDbg 1.09 adds suffixes _1, _2 etc. to file name. Negative side effect of this behaviour is that if you move debugged application to another folder, you will be unable to use old labels and breakpoints. Option "Ignore path and extention" disables name suffixes and ignores path and extention from .udd file when checking for data correctness.

Ignore timestamp

Ignore CRC of code section

When application loads new module, Debugger also compares date of last modification, length of the module and CRC checksum of code section with those listed in the file before restoring user-defined data. This assures some minimal level of security if you have several different DLLs with same name or different versions of the same program. You can disable CRC and timestamp comparisons. The last option is especially useful if you patch executable module.

Save user data outside any module to main .udd file

.UDD files are module-related. If you assign label or comment to, or set breakpoint on address outside any module, OllyDbg by default discards this information when debugging session is finished. If you activate this option, OllyDbg will save this data to the .udd file of main (.exe) module. Note however that Windows does not guarantee that memory will be allocated at the same address each time you run the application.

Allow stepping into unknown commands

When OllyDbg encounters unknown (and possibly invalid) command, it is unable to calculate its length and select appropriate stepping method. If this option is active, OllyDbg assumes that command is neither call nor jump and uses single-step trap to step into this command.

#Allow code injection to get address of WinProc

On NT-based systems, enables code injection to get address of window procedure. Code injection is time-consuming and internally unsafe, especially for multithreading applications.

Warn when breakpoint is outside the code section

Ordinary breakpoint is nothing more than a 1-byte command INT3. When processor executes this command, it generates exception which OllyDbg intercepts. This means that INT3 breakpoint cannot be set on data. When this option is on, OllyDbg warns you if you attempt to set breakpoint outside the code section of loaded modules. If you debug self-extracting file, initialization code frequently resides in a separate section. To avoid frequent messages, you may want to turn this option off.

Warn when terminating active process

To terminate debugged application, OllyDbg calls API function *TerminateProcess()* that shuts down the application instantly, giving it no chance to save data and cleanup. If warning is too annoying to you, check this option.

Warn if not administrator

On NT-based systems, non-administrators have limited rights to debug applications. If you logged in to account without administrative rights, OllyDbg warns you on startup. Check this option to suppress warning.

$#KDebug options

All options

Set high priority when tracing debugged process
Under Windows 95/98, setting high priority for debugged process significantly accelerates processing of debugging events. For example, on Pentium/450 run trace executes 1400 commands per second if priorities of OllyDbg and debugged process are equal, and 2700 when debugged process has higher priority. On NT-based operating systems this option has no noticeable influence on debugging speed (which is anyway higher and reaches 5000 commands per second). In order not to block itself and other programs when debuggee is in free run, OllyDbg increases its priority only when animation or trace (but not forced trace) is active.

Warn about frequent conditional breaks
If you set conditional or memory breakpoint, OllyDbg may get a lot of false debugging events. If condition is not met or memory access violation happened outside the specified area (80x86-compatible processors can protect memory only in chunks of 4096 bytes), OllyDbg automatically continues execution. Debugging events require a lot of attention both from OllyDbg and operating system. Frequent false breaks can make the execution extremely slow (in worst case, less than 2000 commands per second on Pentium/450). If this option is on and number of false breaks exceeds 50 per second, OllyDbg flashes warning message.

Smart update of memory map
If this option is off, OllyDbg walks memory of debugged process on each debugging event. This takes plenty of time and may significantly slow down debugging. When smart update is on, OllyDbg tries to update memory map only when necessary, assuming that drivers or other asynchronous processes don't (re)allocate memory. Option is on by default.

Compress analysis data
Option controls whether analysis data in .udd file is compressed or not. Compression reduces file size but slows down program startup.

Search for references in:
- **Executable code of corresponding module**
- **Memory block currently selected in Disassembler**

Specifies where OllyDbg searches for references.

Use hardware breakpoints to step or trace code
If you step or trace executable code, OllyDbg attempts to use single-step trap. In some cases, for example, if you asked to step over the call to subroutine, or execute string command at once, or run your program till selection, OllyDbg must set temporary breakpoint. Normally it uses command INT3. When OS supports hardware breakpoints, this option is active and there is at least one free hardware breakpoint, OllyDbg sets temporary hardware breakpoint. As inter-process memory access is very time-consuming, hardware breakpoints may accelerate run trace by as much as 20% (trace over). Notice that hardware breakpoints are non-destructive, i.e. don't change the contents of the memory. This feature was proposed by kab.

Hide non-existing source files
Some types of debugging information in Microsoft format include names of source files used to create run-time libraries and similar stuff. Usually they are not available. Check this option to see only source files that really exist.

$#K Events

All options

Make first pause at:

- **System breakpoint**
- **Entry point of main module**
- **WinMain (if location is known)**

When debugger launches Win32 application, OS loads it into memory and calls system INT3 breakpoint. Then it initializes DLLs and calls startup routine. Startup routine performs language-dependent initializations and calls WinMain. This option lets you choose where OllyDbg should stop execution for the first time. Notice that if WinMain is not known, OllyDbg will pause at module entry point.

Break on new module (DLL)
If this option is on, OllyDbg stops execution when debugged application loads new module(s). Unlike some other debuggers, OllyDbg waits until module module is completely loaded into memory, so you can immediately browse it, set breakpoints etc.

Break on module (DLL) unloading
When this option is activated, OllyDbg pauses debugged program when operating system unloads some module. Notice that at this moment module is no longer available for browsing.

Break on new thread
Set this option if you want OllyDbg to stop execution of the debugged program each time it starts new thread.

Break on thread end
When this option is active, OllyDbg stops execution of the debugged program when some thread terminates.

Break on debug string
Set this option if you want OllyDbg to stop execution of the debugged program each time it emits debug string (by call to OutputDebugString).

$#K Exceptions options

All options

#**Ignore memory access violations in KERNEL32**
Windows API includes several functions (*IsBadReadPtr*, *IsBadStringPtr* etc.) that verify whether process has access to the specified range of memory. These functions reside in Kernel32.dll. Internally, they simply try to access memory and return failure when CPU reports memory access violation. If you run program under OllyDbg and this option is off, access violation will be reported to debugger. You can accept it by pressing Shift+F9. If option is on, any memory access violation within Kernel32 will be immediately passed to standard error handler.

Ignore (pass to program) following exceptions:

- **INT3 breaks**
- **Single-step break**
- **Memory access violation**
- **Integer division by 0**
- **Invalid or privileged instruction**
- **All FPU exceptions**

If one of the exception listed above occurs and corresponding option is enabled, OllyDbg passes this exception to the exception handler installed by debugged program. (Alternatively, one can use "shifted" continuation command, like Shift+F7 or Shift+F9 to pass exceptions to exception handler).

Ignore also following custom exceptions or ranges:
Here you can specify custom exceptions or ranges of excptions that should be passed to exception handler. If you want to ignore last detected exception, press Add last exception.

$#K Trace options

All options

Size of run trace buffer
Here you select the size of the circular buffer used to protocol run trace data. Number of records that fits into buffer is very inaccurate and depends on debugged application. Option takes effect only after new run trace buffer is allocated.

Log commands
Asks Run trace to write original commands to the run trace buffer. Useful when you debug self-modified code. Reduces number of commands that fit into buffer by 20-30%.

Show ESP
By default, Run trace window doesn't indicate the actual value of stack pointer. If you want to see it, check this option.

Show flags
By default, Run trace saves flags to the run trace buffer but doesn't display them in Run trace window. Check this option to show flags in window and log file.

#Always trace over system DLLs
When this option is active, OllyDbg doesn't enter system DLLs when animating or tracing into the program. DLL is considered system if it resides in system directory (usually *c:\windows\system* on Windows 95, *c:\winnt\system32* on NT). Main (.exe) file or standalone .dll is user code by definition, even when it resides in system directory. You can toggle system status in Modules window.

Always trace over string commands
When option is selected and OllyDbg animates or traces into the program, it executes string commands with REP prefix at once. Default behaviour is to execute each iteration separately. This option has no influence on Step into command.

#Synchronize CPU and Run trace
When this option is selected and you walk through the run trace, CPU and Run trace windows display the same backward step. On the slower computer, synchronous update may take significant time.

After Executing till RET, step over RET
When option is not active and you select Execute till return (Ctrl+F9), OllyDbg stops on return command. If option is activated, OllyDbg stops after executing return, in most cases on the next command after call.

Group adjacent commands in profile
If option is checked, OllyDbg displays only first command from the group of adjacent commands with same hit count. If option is not checked, OllyDbg displays each command individually.

$#KSFX options

All options

When main module is self-extractable:

Extend code section to include extractor

Many OllyDbg's goodies, like analysis, are available only for the code section of executable module. If this option is active and module is self-extractable, OllyDbg formally extends code section so that it covers self-extractor.

Stop at entry of self-extractor

This is the standard behaviour. OllyDbg doesn't attempt to trace real entry.

Trace real entry blockwise (inaccurate)

OllyDbg uses 4-K blocks to step through the packed code. This method may cause detection of false real entry. For detailed explanation, see here.

Trace real entry bytewise (very slow)

OllyDbg checks every unpacked byte. This method is more reliable than blockwise stepping but very slow. For detailed explanation, see here.

Use real entry from previous run

Once real entry of SFX code is found or set by user, OllyDbg can skip extractor quickly and reliably. This option allows use of known real entry.

Pass exceptions to SFX extractor

This option tells OllyDbg to pass some kinds of software exceptions that occur while tracing for real SFX entry (memory access violation, INT3 breakpoint, division by 0, privileged or illegal instruction) directly to self-extractor.

$#K# String options

All options

Decode Pascal-style string constants

Pascal string consists of byte containing length of the string (0..255 bytes) immediately followed by the string body. It is not necessarily null-terminated. Some high-level languages, like Delphi, make use of Pascal-style constants. When turned on, however, this option increases the chances of misinterpretation.

Dump non-printable ASCII chars as dots

When this option is on, ASCII dump replaces all non-printable characters with dots. The list of characters depends on whether diacritical symbols are allowed or not. When diacriticals are not allowed, printable characters include TAB, CR, LF, space and all codes in range 0x21..0x7E. When diacriticals are allowed, set of printable characters includes also those with diacritical marks (like À, Â, Ã), currency and some special symbols.

Allow diacritical symbols in strings

When OllyDbg attempts to recognize ASCII string and this option is deactivated, string can contain only standard ASCII characters (0x20..0x7E and additionally TAB, CR, LF). Many European languages contain special characters or characters with diacritical marks (like À, Â, Ä, ß). If debugged application supports languages other than English, turn this option on. This, however, increases the number of improperly recognized strings. Default setting is off.

Mode of string decoding:

Plain
Assembler
C

Affects how ASCII and UNICODE strings are displayed in comments. In plain mode, there is no interpretations, all characters are sent to the screen "as is". In Assembler mode, OllyDbg decodes non-printable symbols as hexadecimal numbers or special codes CR, LF or TAB. In C mode, it uses C constructs '\n', '\r', '\x01'.

$#K Address options

All options

Demangle symbolic names

Object-oriented languages, like C++, use technique called name mangling that adds information on types and arguments to symbol. For example, function *Mainwp(HWND *, unsigned int, unsigned int, long)* may be mangled to something like *@Mainwp$qqsp4HWNDuiuil*, which is hardly a readable form. This option tells OllyDbg to demangle symbolic names when they come to the internal table. Changing this option has no influence on the names that were already processed. OllyDbg automatically recognizes mangled names in Borland and Microsoft formats. To decode Microsoft-style mangling, you need *dbghelp.dll* (supplied). Notice that demangled names can be (and often are) ambiguous.

Prepend ordinals to IMPLIB names

OllyDbg can substitute ordinals in DLLs by more meaningful symbolic names extracted from import libraries. If you want to have both ordinal and symbolic name, select this option. For example, some name may have following forms:

As ordinal	#123
As symbol	Symbol
Symbol with prepended ordinal	#123_Symbol

Display address in form of:

HEX, Symbol
Symbol, HEX
Either HEX or Symbol

If some memory address has no symbolic name, OllyDbg displays it as a 8-digit hexadecimal number. If address has symbolic name (label), this option selects one of 3 possible forms:

- 8-digit hexadecimal number followed by symbolic name
- Symbolic name followed by 8-digit hexadecimal number
- Symbolic name only

Show name of local module

When this option is on, OllyDbg prepends module name to symbolic address displayed in the first column of dump or disassembled code.

Highlight symbolic names

When this option is on, OllyDbg highlights symbolic names (labels) displayed in the first column of dump or disassembled code. This can help to understand code.

#By default, sort contents of Names window by:

Address
Name

This option specifies default sort mode for Names windows.

$#K Command options

All options

Use short form of string commands

This option toggles between short form of string-manipulation commands (MOVSD, REP STOSB) and long form (MOVS DWORD PTR ES:[EDI],DWORD PTR DS:[ESI], REP STOS DWORD PTR ES:[EDI]).

Use RET instead of RETN

80x86 processors distinguish between near (RETN) and far (RETF) returns. In most cases, Win32 applications use near form. Toggling this option will display generic RET mnemonics instead of RETN.

Specify size of 16-byte SSE operands as:

DQWORD (Double Quadword)

XMMWORD (eXtended MMX operand)

Here you can specify how OllyDbg decodes size of 128-bit SSE operands in assembler commands.

#**Decode size-sensitive 16/32-bit mnemonics like:**

PUSHA/PUSHAD

PUSHAW/PUSHAD

PUSHAW/PUSHA

80x86 command set includes several commands which mnemonics depend on the size of addresses data. Here you select how Disassembler decodes such commands.

Decode top of stack as:

ST(0)

ST

Both decodings have identical meaning in Assembler commands. Here you can select your preferrable form.

$#K Disasm options

All options

Disassembling syntax:

MASM (Microsoft)
IDEAL (Borland)
HLA (Randall Hyde)

Borland's TASM supports IDEAL mode which (in my humble personal opinion, of course) is much better than MASM. Unfortunately, Microsoft tools don't support it, so for compatibility reasons, programmers usually use quirky MASM syntax. IDEAL syntax requires, among others, that everything that constitutes memory operand must be taken into square brackets.

High Level Assembly language, created by Randall Hyde and makes easy for beginners to write large structured programs in Assembler language, uses yet another, functional syntax where first operand is source. HLA is public domain software, you can download it together with documentation and sources from http://webster.cs.ucr.edu. Compare:

MASM:	MOV EAX,DWORD PTR SS:[EAX+EBP+20]
IDEAL:	MOV EAX,[DWORD SS:EAX+EBP+20]
HLA:	mov([type dword ss:eax+ebp+20], eax);

Assembler allows HLA commands only in HLA mode. By default, OllyDbg uses MASM syntax.

Disassemble in lowercase
When this option is on, commands, register names and modifiers in disassembled commands are displayed in lowercase. When option is off, they are displayed in uppercase. This option has no influence on labels or constants.

Tab between mnemonics and arguments
When option is on, OllyDbg aligns command arguments so that they begin in 8th column.

Extra space between arguments
When option is on, OllyDbg separates command operands by comma, followed by space. When option is off, there is no space between operands

Show default segments
80x86 commands always use some segment (selector) register when accessing memory. When segment register differs from default for the command, one must prepend command with segment-modification prefix. If Show default segments option is on, disassembled address always includes the name of segment register. If option is off, segment register is displayed only when it differs from default. Notice that under Win32 all default segments have base 0 and point to the same memory.

Always show size of memory operands
When this option is on, disassembled memory address always includes data size modifier (BYTE, WORD, DWORD etc.) When option is not set, data size modifier is displayed only when it is not trivial or not obvious from the command. For example, size of memory operand in command MOV [ESI],EAX is 4 byte because EAX is 4 bytes long, but in MOV [ESI],1 addressed memory can be 1, 2 or 4 bytes long.

Show NEAR jump modifiers
When set, tells OllyDbg to add NEAR destination address modifier to all intrasegment indirect jumps or calls. Far jumps and calls are unusual in flat Win32 model.

Show local module name
When this option is on, Disassembler prepends name of destination module to all symbolic or constant addresses. This can help you to understand code.

Show symbolic addresses
When this option is off, Disassembler always decodes constants as hexadecimal numbers. When this option is on and constant has symbolic name, Disassembler displays symbolic name.

$#K CPU options

All options

Synchronize source with CPU
If option is selected, source window automatically follows selection in CPU Disassembler.

Underline fixups
Almost all DLLs and some programs contain fixups that allow to load module on different base address in memory. When Underline fixups option is on, CPU Disassembler and CPU Dump underline fixuped bytes.

Show direction of jumps
When this option is checked, hexadecimal dump of analyzed code displays small arrows directed toward jump or call destination, or short horizontal line if jump destination belongs to different program module.

Show jump path
Show grayed path if jump is not taken
Show jumps to selected command
When jump path option is on, first selected command is a jump and this jump really takes place, OllyDbg draws thin arrow from selection to jump destination. I call this arrow a "jump path". By default, jump path is drawn using highlighting color (red in Black on white color scheme). Optionally, you can gray jump path if jump is not taken. If you ask to show jumps to selected command and this command is a recognized jump destination, OllyDbg draws lines from all jumps to this command.

Center 'Follow'-ed command
When this option is on, following address from any other window in CPU Disassembler pane will show command roughly in the middle of the Disassembler window. Otherwise, followed command appears first in Disassembler.

Number of visible lines after stepped command
Default
1 line
...
10 lines
If you are stepping through the code, this option specifies how many lines remain visible in Disassembler window after current command.

Letter key in Disassembler starts:
New label
Assembler
New comment
Object of last selected type
When you type in some text in Disassembler window, it starts new label, assembler command or comment depending on this setting.

$#K# Register options

All options

Decode registers for any IP

Registers are the most frequently used and modified processor resource. In most cases it makes no sense to decode contents of registers or address expressions which use registers if command is not the next to be executed (i.e. has address equal to the contents of EIP). However if you want to predict program's behavior several steps ahead, you may want to set this option on and track all register modifications by yourself.

#**Automatical FPU/MMX/3DNow! registers**

When this option is off, you must set mode of FPU register decoding (FPU, MMX or 3DNow!) manually. When option is on and program stops on command of one of the listed types, OllyDbg automatically sets FPU decoding mode to type of the command.

Decode SSE registers

If option is on, OllyDbg displays 128-bit SSE registers of actual thread in Registers pane of CPU window and decodes value of registers in the Information pane. Notice that this operation is slow and requires injection of small programs into the debugged application. **Do not use this option unless absolutely necessary!**

Show last error

Turn this option on if you want to display last error detected by thread (as returned by *GetLastError*) in Registers pane and save it to Run trace log. This option slows down the tracing speed by up to 20%.

$#K# Analysis options – part 1

All options

Procedure recognition:

Strict
Heuristical
Fuzzy

Defines how Analyzer recognizes procedures. Strict procedure, by definition, is a continuous piece of code with one defined entry point and at least one return. Heuristical mode allows also procedures with assumed entry. If you select fuzzy mode, any more or less consistent piece of code will be considered separate procedure. Fuzzy mode is especially useful if compiler performs global code optimizations. The probability of misinterpretation, however, is rather high, and I do not recommend fuzzy mode in conjunction with the hit trace.

Show ARGs and LOCALs in procedures

If this option is on and you have analyzed the code, OllyDbg displays addresses like [SS:EBP+8] and [SS:EBP-8] within recognized procedures as [ARG.1] and [LOCAL.2], hinting you that first address is the first argument of the procedure and another is the second word of the local data allocated on stack. Highly optimizing compilers and hand-coded programs, however, may use EBP differently. Indexes are decimal 1-based, which is an exception from general OllyDbg rules. By default, this option is off.

Auto start analysis of main module

When option is active and you start some application for the first time, OllyDbg automatically analyzes code of main module.

Recognize loops and switches

Check this option on to activate recognition of switches and loops.

Decode cascaded IFs as switches

Activate to let Analyzer recognize switches that in reality may be the sequence of cascaded IF operators.

Decode tricky code sequences

Currently OllyDbg recognizes one trick: when option is on, it correctly recognizes CALLs immediately followed by ASCII strings as PUSHes combined with JMPs, like in the following sequence:

```
    PUSH 0
    CALL @1
    DB 'title',0
@1:     CALL @2
    DB 'message',0
@2:     PUSH 0
    CALL MessageBoxA
```

which is a valid call to *MessageBox(0,"message","title",0)*. When option is off, OllyDbg will attempt to interprete strings as ccommands. This trick was reported by BlackArt.

Gray commands that fill gaps between procedures

Analyzer is able to recognize meaningless commands used to fill gaps between aligned procedures. If option is active, they are grayed in the Disassembler window.

Keep analysis between sessions

When this option is on, OllyDbg saves analysis data to the user data file and automatically restores each time this module is opened. This spares time but significantly increases size of module data files on the disk. Switch this option off if you start OllyDbg from the floppy disk.

$#K# Analysis options – part 2

All options

Show arguments of known functions

OllyDbg contains descriptions of about 1900 standard API and 400 C functions. Analyzer recognizes them and comments their arguments. Additionally it marks function scopes, i.e. area where function's arguments are pushed to the stack.

Guess number of arguments of unknown functions

When Analyzer encounters call to unknown function and this option is set, it tries to determine number of doublewords that calling procedure pushes onto the stack and marks them as arguments Arg1, Arg2 and so on. Notice that register arguments, if any, remain unrecognized and are not included in this count. Analyzer goes a secure way. For example, it doesn't recognize procedures without parameters or cases where POP may restore registers before return instead of discarding arguments. Nevertheless, the number of recognized functions is usually very high, strongly improving the readability of code.

Trace contents of registers:

- **Off**
- **In linear sequences**
- **In recognized procedures**

This option specifies whether and how Analyzer predicts contents of registers. If you select **In linear sequences**, Analyzer traces contents of integer registers within the linear pieces of code (i.e. without jumps or calls targeting on it). If you choose **In recognized procedures**, Analyzer recursively combines predictions from all jumps to same target. This last method is time-consuming but gives better results, especially in code produced by optimizing compilers. OllyDbg uses predictions to decode calls to known functions and their arguments. Traced arguments are marked with symbol => (see example).

Use this tracing option at your own risk:

- **Unknown functions preserve EBX, ESI and EDI**

Most API functions preserve registers mentioned above. Analyzer uses this fact when tracing registers. If function is not in the list of known functions and option is unchecked, tracer invalidates EBX, ESI and EDI. By activating this option, you inform Analyzer that all functions follow standard register conventions. In some cases (especially for manually optimized code written in Assembler) this may lead to invalid predictions. This option has no influence on debugging.

$#K Analysis options – part 3

All options
There are many 80x86 commands that can hardly appear in valid program running under Win32. Usually Analyzer comments such command as potentially invalid and treats it as binary data rather than code. You can select here which kinds of potentially invalid commands you want to accept. See also Pause run trace.

Far calls and returns
Far (intersegment or interprivilege or task-switching) calls and returns usually don't appear in Win32 programs.

Modifications of segment registers
Win32 programs usually don't play with segment registers.

Privileged commands
Privileged commands, like HLT or LGDT, are not allowed in user code.

I/O commands
Windows NT does not allow applications to access I/O ports directly. Some drivers, however, can modify I/O Permission Bit Map to allow such access. Windows 95/98 is not so strict. Set this option on if you suspect that debugged program may access I/O ports directly.

Commands that are equivalent to NOP
Compilers usually avoid to include dummy commands into the code. Set option to allow commands like NOP, XCHG EDX,EDX or LEA EAX,[EAX] in the code.

Shifts out of range 1..31
Before performing shifts, 80x86-compatible processors mask shift count with 0x1F, effectively limiting count range to 0..31. Zero count means NOP and hardly will be generated by compiler.

Superfluous prefixes
Set this option on if you want to enable commands with repeated or contradictory prefixes (like MOV EAX,ES:FS:[ESI]), or commands with memory-relevant prefix that do not access memory (ES: XOR EAX,EAX).

LOCK prefixes
Applications itself usually don't make use of LOCK prefix. Instead, they rely on system functions like EnterCriticalSection etc.

Unaligned stack operations
Win32 programs usually keep stack DWORD-aligned. Commands like INC ESP or ADD ESP,7 are seldom in applications.

Non-standard command forms
Some commands, like SETZ, don't use Reg part of ModRegRM address byte. Intel's manuals require that Reg is set to 0, but all available processors ignore Reg when executing such commands. This may change for the future processors.

VxD calls
DOS-based Windows 95 and 98 use VxD calls for communication with virtual devices. Basically, this call is an INT 20 followed by doubleword containing code of requested service. Applications don't call VxD directly, but attempt to disassemble driver file would lead to ununderstandable garbage. This option is by default on for Windows 95 and 98.

$#K# Stack options

All options

Trace stack frames
If this option is on, OllyDbg attempts to trace stack frames created by PUSH EBP; MOV EBP,ESP or ENTER xx,0. Each found frame is marked with a bracket to the left from stack data. It can easily happen that OllyDbg recognizes false frames, for example, remnants from the previous calls. Modern optimizing compilers can generate code that does not produce standard stack frames.

Use stack frame scopes in procedures
If option is on and EBP points to valid stack frame, Stack will decode arguments of known functions.

Highlight RETURNs
When option is selected, OllyDbg highlights addresses in stack that are recognized as returns. As with stack frames, return addresses are often artefacts from the previous calls, the final decision whether address is valid or not belongs to you. OllyDbg does not recognize synthetic calls (PUSH retaddr, JMP entrypoint).

Show arguments of known functions
When this option is on and EIP contains address of call or jump to the known function, or of its entry point, OllyDbg decodes arguments of known function in stack window.

Show names of local variables
If debugging information is available and this option is on, OllyDbg displays names of arguments and local variables of the current function on the stack.

$#K Just-in-time debugging

All options

When some program generates exception, Windows can call registered debugger and attach it to the program. This feature is called Just-in-time debugging.

Some JIT debuggers cautiously stop at system breakpoint. OllyDbg continues execution until it reaches command that generated exception. This is more convenient but causes one problem under Windows 95-based systems: OllyDbg is unable to distinguish between system breakpoint and call to *DebugBreak()*. As a results, it continues execution of the application that paused on *DebugBreak()*. This problem doesn't occur on Windows NT/2000.

Just-in-time debugging is controlled by two entries in the system Registry:

- **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug\Debugger** - contains command that invokes debugger;
- **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug\Auto** - specifies whether to ask user before attaching.

Curiously, "Windows NT" applies to Windows95-based systems, too.

To make OllyDbg just-in-time debugger, press "Make OllyDbg just-in-time debugger"

To restore old setting, press "Restore old just-in-time debugger"

Buttons "Confirm before attaching" and "Attach without confirmation" define whether OllyDbg or external debugger will confirm attaching.

$#K Register tracing - an example

Analyzer

This is an example of how Analyzer can trace the contents of registers. The whole procedur is a linear piece of code.

```
004116E1  ì. C8 000000      ENTER 0,0
004116E5  |. BB 22154500    MOV EBX,<JMP.&KERNEL32.GetPrivateProfile>
004116EA  |. BE 1B174100    MOV ESI,OT.0041171B         ;  ASCII "inifile.ini"
004116EF  |. BF 27174100    MOV EDI,OT.00411727         ;  ASCII "Key1"
004116F4  |. B9 31174100    MOV ECX,OT.00411731         ;  ASCII "SectionName"
004116F9  |. 33C0           XOR EAX,EAX
004116FB  |. 51             PUSH ECX
004116FC  |. 56             PUSH ESI                    ; ì IniFileName => "inifile.ini"
004116FD  |. 50             PUSH EAX                    ; |Default => 0
004116FE  |. 57             PUSH EDI                    ; |Key => "Key1"
004116FF  |. 51             PUSH ECX                    ; |Section => "SectionName"
00411700  |. FFD3           CALL EBX                    ; lGetPrivateProfileIntA
00411702  |. 8B4D 08        MOV ECX,[ARG.1]
00411705  |. 8901           MOV [DWORD DS:ECX],EAX
00411707  |. 59             POP ECX
00411708  |. 83C7 05        ADD EDI,5
0041170B  |. 83C8 FF        OR EAX,FFFFFFFF
0041170E  |. 56             PUSH ESI                    ; ì IniFileName => "inifile.ini"
0041170F  |. 50             PUSH EAX                    ; |Default => FFFFFFFF (-1.)
00411710  |. 57             PUSH EDI                    ; |Key => "Key2"
00411711  |. 51             PUSH ECX                    ; |Section => "SectionName"
00411712  |. FFD3           CALL EBX                    ; lGetPrivateProfileIntA
00411714  |. 8B4D 0C        MOV ECX,[ARG.2]
00411717  |. 8901           MOV [DWORD DS:ECX],EAX
00411719  |. C9             LEAVE
0041171A  l. C3             RETN
0041171B   . 69 6E 69 66 69>ASCII "inifile.ini",0
00411727   . 4B 65 79 31 00>ASCII "Key1",0
0041172C   . 4B 65 79 32 00>ASCII "Key2",0
00411731   . 53 65 63 74 69>ASCII "SectionName",0
```

Function *GetPrivateProfileIntA* is described as a STDFUNC in the data base, and Analyzer assumes that it restores stack and registers EBX, EBP, ESI, EDI. Contents of ECX is preserved on on stack, so it remains defined. Command ADD EDI,5 advances EDI to string "Key2". ORing EAX with FFFFFFFF sets all bits to 1, thus making it completely defined.

$#K Assembler

Built-in OllyDbg Assembler converts text commands to machine code. Many 80x86 commands have several possible encodings. In this case, Assembler tries to select the shortest form. Assembler and Disassembler share the same decoding table, so if some command can be disassembled, it can be assembled too, with one exception: 16-bit addressing modes are not supported. Assembler also understands simplified and alternate mnemonics, like MOVSB (Disassembler decodes it as MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]) or JNE (decoded as JNZ). Dependless on the options, you can type in assembler commands using either Microsoft's MASM or Borland's IDEAL syntax:

MOV EAX,DWORD PTR SS:[EBX*2][EDI+10.] is equivalent to
MOV EAX,[DWORD SS:EBX*2+EDI+10.]

Assembler interprets all constants as hexadecimal unless they explicitly end with decimal point. Don't forget to precede constant with 0 if first digit is a letter!

Assembler treats semicolon (;) and all following symbols as comment. If you are patching code, this comment will be added to the command.

Assembler automatically decides which register to use as an index and which as a base and handles even such weird cases like

MOV EAX,[ESI*9] (equivalent to MOV EAX,[ESI+ESI*8])

or

MOV EAX,[ESI+EDI+ESI] (equivalent to MOV EAX,[EDI+ESI*2])

Assembler also accepts symbolic names. There are some rules you need to follow:

- OllyDbg does not check that label is unique and takes the first found appearance. To limit search to some module, precede label with module name: JMP Myprog.Firstroutine.
- User-defined labels may contain spaces and special characters. Assembler doesn't understand these labels.
- To use import labels, you must specify both resident and destination modules: JMP Myprog.<&KERNEL32.VirtualFree>.

80x86 command set contains only commands XCHG mem,reg and TST mem,reg, but Assembler supports also XCHG reg,mem and TST reg,mem.

You will be warned if you specify LOCK prefix with a command that doesn't accept it.

#Imprecise commands

When searching for the command, you can specify imprecise Assembler instruction. 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)
FPU	Any FPU register (ST0..ST7)
MMX	Any MMX register (MM0..MM7)

CRX	Any control register (CR0..CR7)
DRX	Any debug register (DR0..DR7)
CONST	Any constant
OFFSET	Same as CONST

and imprecise commands:

Command	Matches
JCC	Any conditional jump (JE, JC, JNGE...)
SETCC	Any conditional set byte (SETE, SETC, SETNGE...)
CMOVCC	Any conditional move (CMOVE, CMOVC, CMOVNGE...)

See also: Search for a sequence of commands.

$#K Disassembler

Disassembler recognizes all standard 80x86, protected, FPU, MMX and 3DNow! instructions (including Athlon extensions to MMX command set). It doesn't recognize ISSI commands, although plans exist to include support in the next versions of OllyDbg. Some obsolete or undocumented commands, like LOADALL, are not included, too.

Disassembler correctly decodes 16-bit addresses. However, it assumes all segments to be 32-bit (segment attribute USE32). This is always true for Portable Executable (PE) files. OllyDbg does not support 16-bit New Executables.

If you are familiar with MASM or TASM, you will have no problems with disassembled code. Some peculiarities, however, do exist. Decoding of the following commands differ from the Intel's standard:

AAD (ASCII Adjust AX Before Division) -
Generalized form of this command (non-decimal numbers) is decoded as AAD imm8

AAM (ASCII Adjust AX After Multiply) -
Generalized form of this command (non-decimal numbers) is decoded as AAM imm8

SLDT (Store Local Descriptor Table register) -
Operand is always decoded as 16-bit. 32-bit form of this command stores segment selector in the low-order 16 bits of the destination operand, leaving high-order bits unchanged.

SALC (Sign-extend Carry bit to AL, undocumented) -
OllyDbg supports this undocumented instruction.

PINSRW (Insert Word From Integer Register, Athlon extension to MMX) -
In original AMD documentation, memory form of this command uses 16-bit memory operand whereas register form requires 32-bit register but uses only 16 low-order bits. To facilitate handling, Disassembler decodes register as 16-bit. Assembler supports both forms.

CVTPS2PI and CVTTPS2PI (Convert Packed Single-Precision Floating to Packed Doubleword, Convert with Truncation Packed Single-Precision Floating to Packed Doubleword) -
In these commands, first operand is MMX register, second can be either 128-bit XMM register or 64-bit memory location. To facilitate handling, memory operand is decoded as 128-bit double quadword.

Some instructions have mnemonics that depend on operand size:

size-independent form	explicit 16-bit form	explicit 32-bit form
PUSHA	**PUSHAW**	**PUSHAD**
POPA	**POPAW**	**POPAD**
LOOP	**LOOPW**	**LOOPD**
LOOPE	**LOOPWE**	**LOOPDE**
LOOPNE	**LOOPWNE**	**LOOPDNE**
PUSHF	**PUSHFW**	**PUSHFD**
POPF	**POPFW**	**POPFD**
IRET	**IRETW**	**IRETD**

You can change the decoding of size-sensitive mnemonics. Choose Options, Disassembler and select one of three possible combinations. This option also influences Assembler defaults.

Decoding of MMX and 3DNow! instructions is always enabled, even if host processor doesn't support these instructions.

$#K# Labels

A label is a symbolic name assigned to some address in the linear address space of the debugged application. OllyDbg recognizes four different types of labels:

- **Exports**. These are public symbols accessible by other modules. Each DLL contains at least one export name. OllyDbg adds a special export pseudoname <ModuleEntryPoint>, which indicates the entry point of the module as declared in the PE header. Some exports have no symbolic names, only ordinal numbers. OllyDbg decodes them as #ordinal. If requested in options, OllyDbg prefixes exports with module name.

#- **Imports**. Most executable modules use external symbols defined in other modules. The address of such symbol is not known until referenced module is loaded into memory. To allow Loader resolve external symbols, executable file includes a collection of import tables. For each referenced external symbol, these tables contain name of external module, name of symbol (or its ordinal number) and address of a 32-bit word in the module which will receive address of external symbol. OllyDbg names addresses from import tables by a special pseudonames <&extmod.extsymbol> (or <&extmod.#extordinal> if external symbol is available only by ordinal number). Ampersand '&' here indicates that import name labels not the external symbol itself, rather the pointer to this symbol. Angular brackets '<>' mean that symbol was created by debugger. Notice that import labels are not unique and serve for better understanding the code.

Some compilers implement call to external function as a call to indirect jump to address of external function. OllyDbg decodes this case as <jmp.&extmod.extsymbol>.

- **User-defined labels**. You can assign symbolic name to any address of allocated memory. If address belongs to some module, OllyDbg saves this label to user data file when module unloads and automatically loads next time the module is loaded (even by different program). User labels are not checked for uniqueness and may contain any characters. Among all labels, they have the highest priority. To see all user-defined labels in code sections of currently selected module at once, select Search for|User-defined label in Disassembler. All commands containing your labels will appear in the Reference window, and you can easily browse through the list by pressing Alt+F7 and Alt+F8.

#- **Labels extracted from object files** or libraries by Object Scanner, or **extracted from the debugging information** generated by compiler. These names are marked as "Library" in name list. In actual version OllyDbg cannot guarantee that names extracted by Object Scanner are 100% correct. Library labels have the lowest priority.

$#K# User comments

You can add your own comments to debugged code. If this code belongs to some module, OllyDbg saves comments to user data file when module unloads and automatically loads next time the module is loaded again (even by different program). User comments have priority over automatically generated.

Assembler treats semicolon as start of comment and automatically adds it to the code.

To see all your comments within currently selected module at once, select Search for|User-defined comment in Disassembler. All commands containing your comments will appear in the Reference window, and you can browse through the list by pressing Alt+F7 and Alt+F8.

$#K# References

A reference is an occurrence of a constant in the executable code of some module. OllyDbg allows you to search references to address, to address range, to constant in currently selected command or to some arbitrary constant.

For example, if you specify address or constant 0x401234, the following commands will be recognized as references:

```
MOV EAX,401234
MOV EAX,DWORD PTR [00401234]
MOV BYTE PTR [EBX*4+EDI+00401234],AL
JNE 00401234
CALL 00401234
DD 00401234
```

Analysis not only accelerates search and makes recognition of references more reliable, but also allows to find references in tables.

The complete list of references is displayed in a special Reference window. For your convenience, if you search from Disassembler window, this list will also include the first selected command (highlighted), so you can quickly return back to the place you started from. Reference window also can show the list of found commands, user-defined labels or comments.

If you search for all intermodular calls, all commands or command sequences, switches or referenced text strings, OllyDbg also places results into the Reference window.

In Disassembler, you can browse through the list of references by pressing **Alt+F7** (previous reference) and **Alt+F8** (next reference).

See also: References in Disassembler, Refrences in Dump, User comments.

$#K# Known bugs and problems

1. If OllyDbg is a just-in-time debugger and attaches under Windows 95 to application that executed *DebugBreak()*, this application will start running after attaching. On NT-based systems, application pauses on *DebugBreak()* as expected.

2. Command SMSW (Store Machine Status Word) accepts only AX as argument but compiles it to EAX.

$#K How the Analyzer works - an example

Assume that you have the following piece of code:

```
00440540  55 8B EC 83 C4 F8 DB 45 0C DC 0D B4 05 44 00 83  U......E.....D..
00440550  C4 F8 DD 1C 24 E8 7A 29 01 00 83 C4 08 E8 8A 28  ....$.z).......(
00440560  01 00 89 45 FC 89 45 F8 8B 45 08 83 F8 04 77 28  ...E..E..E....w(
00440570  8A 80 7D 05 44 00 FF 24 85 82 05 44 00 01 02 01  ..}.D..$...D....
00440580  02 01 98 05 44 00 8E 05 44 00 93 05 44 00 8B 45  ....D...D...D..E
00440590  F8 EB 1C 8B 45 FC EB 17 6A 10 68 1A 6D 46 00 68  ....E...j.h.mF.h
004405A0  08 6D 46 00 A1 18 FB 46 00 50 E8 50 4F 01 00 59  .mF....F.P.PO..Y
004405B0  59 5D C3 00 4A D8 12 4D FB 21 09 40 45 72 72 6F  Y]..J..M.!.@Erro
004405C0  72 21 00 49 6E 76 61 6C 69 64 20 70 61 72 61 6D  r!.Invalid param
004405D0  65 74 65 72 00                                   eter.
```

What can it be? Ordinary disassembler will produce the following results:

```
00440540  55               PUSH EBP
00440541  8BEC             MOV EBP,ESP
00440543  83C4 F8          ADD ESP,-8
00440546  DB45 0C          FILD DWORD PTR [EBP+C]
00440549  DC0D B4054400    FMUL QWORD PTR [4405B4]
0044054F  83C4 F8          ADD ESP,-8
00440552  DD1C24           FSTP QWORD PTR [ESP]
00440555  E8 7A290100      CALL 00452ED4
0044055A  83C4 08          ADD ESP,8
0044055D  E8 8A280100      CALL 00452DEC
00440562  8945 FC          MOV DWORD PTR [EBP-4],EAX
00440565  8945 F8          MOV DWORD PTR [EBP-8],EAX
00440568  8B45 08          MOV EAX,DWORD PTR [EBP+8]
0044056B  83F8 04          CMP EAX,4
0044056E  77 28            JA SHORT 00440598
00440570  8A80 7D054400    MOV AL,BYTE PTR [EAX+44057D]
00440576  FF2485 82054400  JMP DWORD PTR [EAX*4+440582]
0044057D  0102             ADD DWORD PTR [EDX],EAX
0044057F  0102             ADD DWORD PTR [EDX],EAX
00440581  0198 0544008E    ADD DWORD PTR [EAX+8E004405],EBX
00440587  05 44009305      ADD EAX,5930044
0044058C  44               INC ESP
0044058D  008B 45F8EB1C    ADD BYTE PTR [EBX+1CEBF845],CL
00440593  8B45 FC          MOV EAX,DWORD PTR [EBP-4]
00440596  EB 17            JMP SHORT 004405AF
00440598  6A 10            PUSH 10
0044059A  68 1A6D4600      PUSH 4405BC
0044059F  68 086D4600      PUSH 4405C3
004405A4  A1 18FB4600      MOV EAX,DWORD PTR [46FB18]
004405A9  50               PUSH EAX
004405AA  E8 504F0100      CALL 004554FF
004405AF  59               POP ECX
004405B0  59               POP ECX
004405B1  5D               POP EBP
004405B2  C3               RETN
004405B3  004A D8          ADD BYTE PTR [EDX-28],CL
004405B6  124D FB          ADC CL,BYTE PTR [EBP-5]
004405B9  2109             AND DWORD PTR [ECX],ECX
004405BB  40               INC EAX
004405BC  45               INC EBP
004405BD  72 72            JB SHORT OT.004600F1
004405BF  6F               OUTS DX,DWORD PTR [EDI]
004405C0  72 21            JB SHORT OT.004600A3
004405C2  0049 6E          ADD BYTE PTR [ECX+6E],CL
004405C5  76 61            JBE SHORT OT.004600E8
004405C7  6C               INS BYTE PTR [EDI],DX
004405C8  696420 70 6172616>IMUL ESP,DWORD PTR [EAX+70],6D617261
004405D0  65:74 65         JE SHORT OT.004600F8
004405D3  72 00            JB SHORT OT.00460095
```

Hmm... Looks like a function... At least it creates standard stack frame (PUSH EBP; MOV EBP,ESP) and returns back. But what happens at address 0044057D? These commands look very suspicious. And what stays after return? Some ASCII text and something more... Let's run code analysis and object scanning. In a minute this code looks much better:

```
Dummy    ì $ 55              PUSH EBP                      ;
00440541 | . 8BEC            MOV EBP,ESP                   ;
00440543 | . 83C4 F8         ADD ESP,-8                    ;
00440546 | . DB45 0C         FILD [ARG.2]                  ;
00440549 | . DC0D B4054400   FMUL QWORD PTR [4405B4]       ;
0044054F | . 83C4 F8         ADD ESP,-8                    ;
00440552 | . DD1C24          FSTP QWORD PTR [ESP]          ;
00440555 | . E8 7A290100     CALL Test._sqrt               ;
0044055A | . 83C4 08         ADD ESP,8                     ;
0044055D | . E8 8A280100     CALL Test.__ftol              ;
00440562 | . 8945 FC         MOV [LOCAL.1],EAX             ;
00440565 | . 8945 F8         MOV [LOCAL.2],EAX             ;
00440568 | . 8B45 08         MOV EAX,[ARG.1]               ;
0044056B | . 83F8 04         CMP EAX,4                     ;
0044056E | . 77 28           JA SHORT Test.00440598        ;
00440570 | . 8A80 7D054400   MOV AL,BYTE PTR [EAX+44057D] ;
00440576 | . FF2485 82054400 JMP DWORD PTR [EAX*4+440582] ;
0044057D | . 01              DB 01                        ;  Index table to switch 00440582
0044057E | . 02              DB 02                        ;
0044057F | . 01              DB 01                        ;
00440580 | . 02              DB 02                        ;
00440581 | . 01              DB 01                        ;
00440582 | . 98054400        DD Test.00440598             ;  Switch table used at 00440576
00440586 | . 8E054400        DD Test.0044058E             ;
0044058A | . 93054400        DD Test.00440593             ;
0044058E | > 8B45 F8         MOV EAX,[LOCAL.2]            ;
00440591 | . EB 1C           JMP SHORT Test.004405AF      ;
00440593 | > 8B45 FC         MOV EAX,[LOCAL.1]            ;
00440596 | . EB 17           JMP SHORT Test.004405AF      ;
00440598 | > 6A 10           PUSH 10                      ; ì fuStyle =
MB_OK|MB_ICONHAND|MB_APPLMODAL
0044059A | . 68 1A6D4600     PUSH Test.004405BC           ; | lpszTitle = "Error!"
0044059F | . 68 086D4600     PUSH Test.004405C3           ; | lpszText = "Invalid parameter"
004405A4 | . A1 18FB4600     MOV EAX,DWORD PTR [46FB18]   ; |
004405A9 | . 50              PUSH EAX                     ; | hwndOwner
004405AA | . E8 504F0100     CALL Test.MessageBoxA        ; ⌊ MessageBoxA
004405AF | > 59              POP ECX                      ;
004405B0 | . 59              POP ECX                      ;
004405B1 | . 5D              POP EBP                      ;
004405B2 ⌊ . C3              RETN                         ;
004405B3     00              DB 00                        ;
004405B4   . 4AD8124DFB210940DQ FLOAT 3.141592600000000   ;
004405BC   . 45 72 72 6F 72 >ASCII "Error!",0             ;
004405C3   . 49 6E 76 61 6C >ASCII "Invalid paramete"     ;
004405D3   . 72 00           ASCII "r",0                  ;
```

See! This is really a complete procedure. Suspicious commands at 0044057D are in reality index and address tables of a two-stage switch! Analyzer had also recognized and decoded a call to a standard API function MessageBox (ASCII version). This call displays modal error message „Invalid parameter" with only one button „OK".

Byte at 004405B3 is used as a filling to align following floating-point constant (guess what does it mean?) to a DWORD boundary. Two text strings follow the floating constant. Object scanner also recognized calls to library functions sqrt and _ftol (the last converts double to long) which reside in different memory region.

For those who are curious about the original source, here it is. I prepared this otherwise useless code to illustrate as many analyzer features as possible:

```
int Dummy(int i,int j) {
  int a[2];
  a[0]=a[1]=sqrt(j*3.1415926);
  switch (i) {
    case 0:
    case 2:
    case 4:
      return a[0];
    case 1:
    case 3:
      return a[1];
    default:
      MessageBox(hwmain,"Invalid parameter","Error!",
      MB_OK|MB_ICONSTOP|MB_APPLMODAL);
    break;
  }
}
```

Notice that MB_ICONSTOP and MB_ICONHAND are synonyms.

$#K Custom function descriptions

#Introduction

OllyDbg includes (as internal resource) names and types of arguments of more than 1900 standard API functions and more than 400 standard C functions. Analyzer uses these descriptions to make debugged program more understandable. Compare, for example, call to *CreateFont* before analysis:

```
PUSH OT.00469F2A               ; ASCII "Times New Roman"
PUSH 12
PUSH 2
PUSH 0
PUSH 0
PUSH 0
PUSH 0
PUSH 0
MOV EAX,DWORD PTR [49FA70]
PUSH EAX
PUSH 190
PUSH 0
PUSH 0
PUSH 0
PUSH 10
CALL <JMP.&GDI32.CreateFontA>
```

and after:

```
MOV EAX,DWORD PTR [49FA70]
PUSH OT.00469F2A               ; ì FaceName = "Times New Roman"
PUSH 12                        ; | PitchAndFamily = VARIABLE_PITCH|FF_ROMAN
PUSH 2                         ; | Quality = PROOF_QUALITY
PUSH 0                         ; | ClipPrecision = CLIP_DEFAULT_PRECIS
PUSH 0                         ; | OutputPrecision = OUT_DEFAULT_PRECIS
PUSH 0                         ; | CharSet = ANSI_CHARSET
PUSH 0                         ; | StrikeOut = FALSE
PUSH 0                         ; | Underline = FALSE
PUSH EAX                       ; | Italic => TRUE
PUSH 190                       ; | Weight = FW_NORMAL
PUSH 0                         ; | Orientation = 0
PUSH 0                         ; | Escapement = 0
PUSH 0                         ; | Width = 0
PUSH 10                        ; | Height = 10 (16.)
CALL <JMP.&GDI32.CreateFontA>  ; ï CreateFontA
```

Of course, second version is definitely easier to understand. API function *CreateFont* has 14 parameters. Analyzer marks all these parameters with their names and decods those of known value. Register tracing was on, so Analyzer also decoded the value of *Italic* as an actual contents of doubleword at address 49FA70. Decoding uses actual values of arguments, so if the contents of [49FA70] will change, *Italic* will change accordingly. OllyDbg also decodes arguments of known function in stack window when EIP points to call or jump to this function or to its entry point.

OllyDbg decodes arguments of printf()-alike functions with variable number of parameters:

```
PUSH EAX                       ; ì <%.*s>
PUSH E8                        ; | <*> = E8 (232.)
PUSH EBX                       ; | <%08X>
```

```
PUSH Mymodule.004801D2          ; |format = "Size %08X (%.*s) bytes"
PUSH ESI                        ; |s
CALL Mymodule.sprintf           ; \sprintf
```

You can define your own functions. Each time you open some application, OllyDbg resets table of function arguments and fills it with embedded descriptions. Then it tries to open files "<OllyDbg directory>\common.arg" and "<OllyDbg directory>\<Application>.arg", where <Application> stays for the 8-byte (DOS) name of the program being debugged (without path and extention).

A simple .arg file may look like this:

```
INFO Simple .ARG file that decodes CreateHatchBrush
TYPE HS_X
  IF 0 "HS_HORIZONTAL"
  IF 1 "HS_VERTICAL"
  IF 2 "HS_FDIAGONAL"
  IF 3 "HS_BDIAGONAL"
  IF 4 "HS_CROSS"
  IF 5 "HS_DIAGCROSS"
  ELSEINT
END
TYPE COLORREF
  IF 0 "<BLACK>"
  IF 00FFFFFF "<WHITE>"
  OTHERWISE
  TEXT "RGB("
  FIELD 000000FF
  UINT
  TEXT ","
  FIELD 0000FF00
  UINT
  TEXT ","
  FIELD 00FF0000
  UINT
  TEXT ")"
END
STDFUNC CreateHatchBrush
  "style" HS_X
  "colorref" COLORREF
END
```

Standard Windows API function *CreateHatchBrush(int style,int colorref)* takes two parameters. First must be the hatch style and the second consists of red, green and blue colour components packed in the lower 3 bytes of a 32-bit integer. To decode such arguments, file defines two new argument types: HS_X and COLORREF.

Hatch style is a simple enumeration, where 0 stays for HS_HORIZONTAL, 1 – HS_VERTICAL etc. IF operator compares argument with the first operand (notice: always hexadecimal!) and, if they coincide, displays text which is a second operand. But what if there is no match? Operator ELSEINT orders OllyDbg to decode argument as an integer.

COLORREF is more complicated. First it tries to decode two widely used colours: black (all components are 0) and white (all are 0xFF). If there were no match, COLORREF tries to decode colour as a structure containing intensities of red, green and blue. FIELD makes logical AND of the argument fith the first operand. Then it shifts both resulting integer and mask to the right synchronously so long that the least significant bit of the mask becomes 1. Then integer is displayed as unsigned decimal. I do it three times, separately for each colour component. TEXT prints its argument unconditionally. If argument is 00030201, COLORREF decodes it as RGB(1.,2.,3.)

Most API functions remove parameters from the stack and preserve the contents of registers EBX, EBP, ESI and EDI. Declaring this function as STDFUNC reports this fact to Analyzer. Otherwise,

describe it as FUNCTION.

And what if argument is a combination of several fields and bit values, like *fdwPitchAndFamily* above? Look at this:

```
TYPE FF_PITCH
  MASK 03
  IF 00 "DEFAULT_PITCH"
  IF 01 "FIXED_PITCH"
  IF 02 "VARIABLE_PITCH"
  ELSEHEX
  TEXT "|"
  MASK 0C
  BIT 04 "4|"
  BIT 08 "8|"
  MASK FFFFFFF0
  IF 00 "FF_DONTCARE"
  IF 10 "FF_ROMAN"
  IF 20 "FF_SWISS"
  IF 30 "FF_MODERN"
  IF 40 "FF_SCRIPT"
  IF 50 "FF_DECORATIVE"
  ELSEHEX
END
```

First two bits contain type of pitch and must be decoded together. I extract these bits using MASK 03 and decode with a sequence of IFs. I add concatenation symbol, extract bits 2 and 3 and decode them separately. Then I extract the rest and decode known combinations.

OllyDbg removes concatenations ('|'), spaces, commas, semicolons and equal signs from the end of the generated string.

Actual version of analyzer can decode only 32-bit arguments. For example, you can't decode double or long double function parameters.

#Formal description

Custom decoding information consists of function and type descriptors. Function descriptor is rather straightforward:

FUNCTION|STDFUNC [modulename.]functionname
  <Name of first argument> <Type of first argument>
  ...
  <Name of last argument> <Type of last argument>
END

Use keyword STDFUNC if function removes arguments from stack and preserves registers EBX, EBP, ESI and EDI. Most API functions follow these rules. Declare it as FUNCTION in all other cases. Name of module (EXE or DLL) is optional. If it is absent, OllyDbg matches function with given name in any module. Name of module is not case-sensitive.

Name of function is always case-sensitive. UNICODE-enabled functions *must* include A or W suffix, like in *SetWindowTextA*.

The order of arguments corresponds to C-style parameter passing convention. Contrary to 16-bit Windows, 32-bit API functions use this convention too. If name of argument consists of several words or includes special characters, include it in doble quotes ("). Ellipsis (...) is a special record indicating variable number of arguments, as in C language. It must be last in the description of the function. OllyDbg does not try to decode such arguments. Function with empty argument list

is treated as functionname(void).

OllyDbg supports only 32-bit arguments. Some argument types are predefined:

INT	value displayed both in hex and signed integer formats
UINT	value displayed both in hex and unsigned integer formats
HEX	value displayed in hexadecimal format
BOOL	TRUE or FALSE
CHAR	ASCII character
WCHAR	UNICODE character
FLOAT	32-bit floating-point number
ERRCODE	system error code (like reported by *GetLastError*())
ADDR, **PTR**	address (special case: NULL)
ASCII	pointer to ASCII string
UNICODE	pointer to UNICODE string
FORMAT	ASCII format string used in *printf*()-alike functions - don't use it for *scanf*()!
WFORMAT	UNICODE format string used in *wsprintfW*()-alike functions - don't use it for *wscanfW*()!
RECT	pointer to RECT structure (rectangle)
MESSAGE	pointer to MSG structure (ASCII Windows message)
WMESSAGE	pointer to MSG structure (UNICODE Windows message)
HANDLE	handle (special cases: NULL, ERROR_INVALID_HANDLE)
HWND	handle of window
HMODULE	handle of module
RSRC_STRING	resource string with given index
NULL, **DUMMY**	argument is present but skipped from decoding

You can't redefine predefined type. Custom type definition allows to split argument in several fields and decode them separatly. Type descriptor has following format:

TYPE typename
  [**TEXT** "anytext"]
  [<field selector>]
  <field decoding>
  <field decoding>
  [**TEXT** "anytext"]
  [**PURGE**]
  ...
  <field selector>
  <field decoding>
  <field decoding>
  [**TEXT** "anytext"]
END

Length of typename is limited to 16 characters. OllyDbg adds "anytext" to the generated decode string unconditionally. Field selectors extract part of the argument for decoding. All field decoders that follow selector use extracted field:

MASK hexmask - field is argument AND hexmask;

FIELD hexmask - field is argument AND hexmask, then OllyDbg shifts field and hexmask together to the right so that the least significant bit "1" in mask comes in position 0. For example, if argument is 0xC250, FIELD F0 extracts 5.

SIGFIELD hexmask - field is argument AND hexmask. OllyDbg shifts field and hexmask

together to the right so that the least significant bit "1" in mask comes in position 0 and sign-extends field. For example, if argument is 0xC250, SIGFIELD FF00 extracts 0xFFFFFFC2.

Simple field decoders display contents of the whole field at once:

HEX - displays contents of the field in hexadecimal form;

INT - displays contents of the field as signed decimal number (with decimal point);

UINT - displays contents of the field as unsigned decimal number (with decimal point).

CHAR - displays contents of the field as ASCII character.

If field is an enumeration, use sequence of IFs followed if necessary by TRYxxx and/or ELSExxx:

IF hexvalue "text" - if field is equal to hexvalue, add text to the output string;

TRYASCII - if field looks like a pointer to ASCII string, display string;

TRYUNICODE - if field looks like a pointer to UNICODE string, display string;

TRYORDINAL - if field looks like an ordinal (16 most significant bits are 0) display ordinal (integer preceded by '#';

OTHERWISE - if some previous IF succeeded, terminates decoding; otherwise, decoding continues;

ELSEINT - if all previous IFs and TRYxxx's were unsuccessful, display field as signed decimal (with decimal point);

ELSEHEX - if all previous IFs and TRYxxx's were unsuccessful, display field in hexadecimal form;

ELSECHAR - if all previous IFs and TRYxxx's were unsuccessful, display field as ASCII character;

ELSEWCHAR - if all previous IFs and TRYxxx's were unsuccessful, display field as wide (UNICODE) character.

If field is a set of bits, use sequence of BITs followed if necessary by BITZ and BITHEX:

BIT hexmask "text" - if value AND hexmask is not 0, add text to the output string;

BITZ hexmask "text" - if value AND hexmask is 0, add text to the output string;

BITHEX hexmask - if value AND hexmask is not 0, display value AND hexmask in hexadecimal form.

Special operator **PURGE** removes following trailing symbols from the output string:

Space	' '
Comma	','
OR	'|'
Colon	':'
Equals	'='

This facilitates in some cases the decoding. Operator **END** marks end of type definition and automatically executes PURGE.

#Precompiled types

OllyDbg includes, as precompiled resource, the descriptions of more than 150 types. Some of them are listed below. You can use these types in your custom files:

LANG_X - Windows language IDs (0 - neutral, 9 - English, C- French etc.)

GENERIC_X - type of access (GENERIC_READ, GENERIC_WRITE...)

FILE_SHARE_X - type of file sharing (FILE_SHARE_READ, FILE_SHARE_WRITE)

CREATEFILE_X - mode of file creation (CREATE_NEW, OPEN_EXISTING...)

FILE_ATTRIBUTE_X - file attributes (READONLY, SYSTEM, DELETE_ON_CLOSE...)

RT_AXX - resource type (RT_CURSOR, RT_GROUP_ICON, ASCII string...)

RT_WXX - resource type (RT_CURSOR, RT_GROUP_ICON, UNICODE string...)

COORD - COORD structure "(X=xxx,Y=yyy)"

STD_IO_X - standard handles (STD_INPUT_HANDLE, STD_ERROR_HANDLE...)

GMEM_X - type of global memory (GMEM_FIXED, GPTR...)

LMEM_X - type of local memory (LMEM_FIXED, LPTR...)

FSEEK_X - file seek type (FILE_BEGIN, FILE_CURRENT...)

OF_X - file mode (OF_READ, OF_SHARE_COMPAT, OF_VERIFY...)

O_X - file creation mode (O_RDONLY, O_BINARY, SH_COMPAT...)

SEMAPHORE_X - type of semaphore (SEMAPHORE_ALL_ACCESS, SYNCHRONIZE...)

SLEEP_TIMEOUT - timeout (INFINITE or time)

ROP - some standard ROP codes (SRCCOPY, MERGEPAINT...)

COLORREF - RGB color values ("<WHITE>", "RGB(rr.,gg.,bb.)"...)

WS_X - window style (WS_OVERLAPPED, WS_POPUP...)

WS_EX_X - extended window style (WS_EX_DLGMODALFRAME, WS_EX_TOPMOST...)

MF_X - menu flags (MF_BYPOSITION, MF_ENABLED...)

WM_X - Windows ASCII message type (WM_CREATE, WM_KILLFOCUS, CB_SETCURSEL...)

WM_W - Windows UNICODE message type (WM_CREATE, WM_KILLFOCUS,

CB_SETCURSEL...)

VK_X - virtual key code (VK_LBUTTON, VK_TAB, VK_F10...)

MB_X - message box style (MB_OK, MB_ICONHAND...)

HKEY_X - predefined registry handles (HKEY_CLASSES_ROOT, HKEY_LOCAL_MACHINE...)

There are more precompiled types. In general, if constants are defined as ABC_xxxxxxxx in header file, try precompiled type ABC_X.

$#K# Search

OllyDbg allows you to search for:

symbolic name (label)
binary string
constant
command
sequence of commands
intremodular calls
modified command or data
user-defined label
user-defined comment
text string
record in run trace
referencing commands

$#K Names window

Names window lists all symbolic names (labels) in some module. These names include exports (public symbols accessible by other modules), imports (references to external symbols from other modules), user-defined labels and labels extracted from debugging information or object files. To view names, press Ctrl+N in Disassembler or Executable Modules.

Imports are preceeded by the name of module where external symbols are located. Some modules may export symbols by ordinal numbers. OllyDbg displays ordinals in form #nnn, where nnn is the decimal number.

You can set default order of names (sorted by address or alphabetically) in Address options. To change sorting order, press corresponding button in bar or select sorting criterium from the pop-up menu. If you sort names alphabetically, please notice that OllyDbg ignores module names and some leading symbols like underscore, question mark or ampersand.

To find some name, simply start typing it on the keyboard. Title bar of the Names window displays characters you tipped so far, and the context of window scrolls to the first matching name. Comparison is case-insensitive and also ignores module names and leading non-letter characters. Backspace deletes last entered character.

Names window supports following actions:

Actualize - updates list of names associated with the module. OllyDbg automatically updates list if some name within the module is added, deleted or modified, but doesn't trace changes in different modules.

Follow in Disassembler (Enter) - follows address in the Disassembler window. This menu entry is available only if address points to executable code of the current module.

Follow import in Disassembler - Import is a pointer to named address in external module. This item follows contents of a doubleword at associated address in the Disassembler window.

Follow in Dump - follows address in the CPU Dump.

Find references - searches for all commands in the module that reference name.

Find references to import (Enter) - searches for all commands in the module that reference address of import. Some compilers implements calls to imported function as calls to indirect jump to imported function. If only one such jump exists, OllyDbg automatically searches for references to this jump. Notice that that this option and "Follow in Disassembler" are mutually exclusive.

View call tree - displays call tree for selected name.

Help on symbolic name (Ctrl+F1) - if API help file is selected, searches for the explanation of symbolic name.

Toggle breakpoint (F2) - toggles INT3 breakpoint on the command at given address.

Conditional breakpoint (Shift+F2) - allows to set conditional breakpoint on the command at given address.

Conditional log breakpoint (Shift+F4) - allows to set logging breakpoint. For more details, see Breakpoints.

Toggle breakpoint on import - toggles INT3 breakpoint on the command pointed to by the doubleword at specified address.

Conditional breakpoint on import - allows to set conditional breakpoint on the command pointed to by the doubleword at given address.

Conditional log breakpoint on import - allows to set logging breakpoint on import.

Set breakpoint on every reference - searches for references and sets unconditional INT3 breakpoint on every reference. This command is very useful if you want to break on all calls to some API function under Windows 95-based OS. Under Windows NT, you can set breakpoint directly on the entry to API function. This command doesn't attempt to find references to functions loaded by *GetProcAddress*.

Set log breakpoint on every reference - allows you to search for references and set logging breakpoint with the same conditions on every reference.

Remove all breakpoints - searches for references and removes any kinds of breakpoints from all found references.

Appearance - see here.

$#KSSE support

SSE (Streaming SIMD Extentions) instructions were first introduced by Intel for Pentium III processor. They allow for simultaneous processing of up to 4 single-precision floating-point operands. OllyDbg 1.06 disassembles and assembles SSE commands and can display and modify the contents of 128-bit SSE registers. This task is not as trivial as it looks at the first glance. The main problem is the lack of debugging support in all actual versions of Windows, that is, Debugger can't get these registers from the OS. To overcome this limitation, OllyDbg injects and executes small piece of code that reads or updates SSE registers within the debugged thread. This operation is slow and dangerous, and I recommend to disable decoding of SSE registers unless absolutely necessary. Run trace doesn't save SSE registers to the log.

Notice that SSE2 (Pentium 4 extentions) is not supported.

$#KSearch for binary strings

Other types of search

OllyDbg can search for binary strings up to 256 bytes in length. In most windows, this function has shortcut **Ctrl+B**.

You can enter search pattern in ASCII, UNICODE or hexadecimal format, as shown on the picture above. For your convenience, the edit controls are synchronized. Any changes in one control are immediately displayed in two remaining. By pressing **Ctrl+ArrowUp** or **Ctrl+ArrowDn**, you can quickly change to the same byte location in different control. Two buttons with arrows to the left and to the right select one of the last 10 entered patterns.

HEX control allows you to exclude single nibbles or bytes from the comparison. Type question mark (**?**) in HEX control to mark nibbles as masked. If you paste to the HEX control, OllyDbg scans text on clipboard and extracts hexadecimal digits (0..9, A..F, a..f) and question marks (**?**), ignoring all other symbols. Additionally, you can make search case-insensitive.

Sometimes you may need to locate some piece of code in the different version of debugged program. If code contains no relocations, you can select it, make binary copy to clipboard, open another version and paste the contents of clipboard to the HEX control of search window. Another option (binary copy with masked fixups) replaces fixups with question marks, creating search patterns that are insensitive to the load address.

$#K Search for a sequence of commands

Other types of search

Compilers usually use identical sequences of commands to perform identical tasks. For example, many procedures create stack frame by executing sequence PUSH EBP; MOV EBP,ESP. To locate entry points, one may attempt to search for a binary code 55, 8B, EC. However, there are two problems: command MOV EBP,ESP has also different encoding 89, E5, and compilers that optimize for a Pentium-class processor may insert intermediate commands that don't affect EBP and ESP.

Another example: to access an array of structures that are 100 bytes long, compiler may generate following code:

```
LEA EAX,[4*EAX+EAX]              ; Multiplies EAX by 5
LEA EAX,[4*EAX+EAX]              ; Multiplies EAX by 5
MOV EBX,[4*EAX+base_of_array]    ; Multiplies EAX by 4
```

or slightly different code that leads to the same result:

```
LEA ECX,[4*EAX+EAX]
LEA ECX,[4*ECX+ECX]
MOV EBX,[4*ECX+base_of_array]
```

or yet another of the hundreds of different possibilities.

To help you in these and similar cases, OllyDbg implements a powerful search for a sequence of commands (shortcur **Ctrl+S** in Disassembler window). You can enter a sequence of up to 8 assembler commands. For every listed command, OllyDbg generates all possible encodings and during the search tests for every possible combination of these commands. You can use imprecise operands and commands. However, command LEA R32,[4*R32+R32] matches also LEA ESI,[EAX*4+EBX] - not exactly what you want. OllyDbg allows you to specify matching pseudoregisters RA and RB. Like R32, they are substitutes for any general-purpose 32-bit register, but they are always the same within the whole sequence. RA and RB may coincide. Thus, LEA RA,[4*RA+RA] will match LEA EAX,[4*EAX+EAX] and LEA ESI,[4*ESI+ESI], but not LEA ESI,[EAX*4+EBX].

To skip up to n ambiguous commands, use construct ANY n. If you expect up to 2 intermediate commands in the header of the procedure, use

```
PUSH EBP
ANY 2
MOV EBP,ESP,
```

and possible search pattern in the second case is, for example,

```
LEA RB,[4*RA+RA]
ANY
LEA R32,[4*RB+RB]
```

As for any other kind of search, press **Ctrl+L** to search for another occurence of the sequence. You can also find all sequences at once and then walk this list using keyboard shortcuts **Alt+F7** and **Alt+F8**.

$#KSearch for modified commands or data

Other types of search

Dump windows support backup copy of data. OllyDbg creates backup automatically if you modify code in Disassembler window or data in CPU Dump. Whenever there is a difference between actual code or data and backup, OllyDbg highlights it. You can find next difference by selecting Search for|Modified code or Search for|Modified data from the pop-up menu. As for any other kind of search, press **Ctrl+L** to search for another modification.

$#K# Profile

Profile always works together with the Run trace. Profile simply calculates how many times the address of some command appears in the run trace log. This operation may take significant time, so Profile window updates only on your explicit request. If several consecutive commands have the same count, Profile window displays only the first command in a sequence.

Profile window supports following actions:

Actualize - recalculates profile for selected module;

Follow in Disassembler (Enter) - follows command or sequence of commands with identical count in Disassembler;

Find in Run trace - finds last (most recent) occurence of the command in the Run trace. If option Synchronize CPU and Run trace is active, CPU window displays command with original register values.

$#K Call stack

Call stack window (shortcut **Alt+K**) attempts to backtrace the sequence of calls on the stack of selected thread and displays it, together with known or suggested parameters of called functions. This task is easy when called functions create standard stack frames (PUSH EBP; MOV EBP,ESP). Modern optimizing compilers don't bother about stack frames, so OllyDbg uses different tricks and suggestions. For example, it tries to trace code to the next return and counts all pushes, pops or ESP modifications. If this doesn't succeed, it attempts different, more risky and time-consuming approach: it walks stack, searches for all possible return addresses and checks whether function called in preceeding operator includes analyzed command. There are also other, rather questionable heuristics. Stack walk can be really slow. OllyDbg makes it only if call stack window is open.

Call stack window contains 5 columns: Address, Stack, Procedure, Called from, Frame. **Address** column contains address on the stack, **Stack** displays value of the corresponding return address or argument.

Procedure (or **Procedure / arguments**) is where call stack displays address of called function. In some cases OllyDbg is not sure whether this address is correct and adds one of the following markers:

?	Found entry point is not reliable
Maybe	OllyDbg was unable to find exact entry point, reported address is guessed using heuristics
Includes	OllyDbg was unable to find entry point and only knows that this procedure includes displayed address

By pressing the bar button or selecting "Hide/Show arguments" from menu, you can toggle on or off function arguments.

Called from is the address of the command that called this procedure. The last column, **Frame**, is by default hidden and displays value of frame pointer (register EBP), if known.

Stack walk is more reliable and significantly faster when called functions are analyzed.

See also: Call tree

$#K# Call tree

Call tree (shortcut **Ctrl+K** in Disassembler) uses results of analysis to prepare list of functions called by given procedure, directly or indirectly, and list of all known calls to the given function. As a nice side effect, it recognizes whether selected procedure is explicitly recursive. 'Explicitly' means that it can't trace calls with unknown destination, like CALL EAX. If procedure makes unknown calls, Call tree adds marker "Unknown destination(s)".

Some of the called functions are commented with one of the following words:

Leaf	Calls no other functions
Pure	Calls no functions, has no side effects
RETN	Consists of single command RETN
Sys	Function in system DLL. By definition, system DLL is DLL residing in system directory

To walk the call tree, doubleclick address in "Called from" or "Calls/Calls directly" column. Call tree window keeps the history of your walk (shortcuts '**-**' and '**+**').

If debugged program consists of several modules, I recommend to analyze all these modules. Call tree doesn't try to process system functions.

See also: Analysis, Call stack

$#K Windows

This window displays list of all windows owned by debugged application and their most important parameters.

NT-based systems hide address of window procedure (function that processes all messages to the window). This effectively disables subclassing of windows belonging to different process but creates problems for debugger. OllyDbg can read address of window procedure by executing *GetWindowLong* in the context of debugged application using code injection. This method is time-consuming and sometimes (especially for multithreaded applications) unsafe. I use it only when debugged application is paused and injection is enabled by option Allow code injection to get address of WinProc.

Due to the lack of place, many columns in this window contain additional information that is normally hidden in the columns. For example, column WinProc marks subclassed windows (window procedure is not the same as class procedure), column ID displays ID of the control in decimal form and Style decodes window's style as a set of WM_xxx flags. To view additional information, change width of corresponding column.

You can set breakpoint on window message, for example WM_PAINT, or group, like mouse messages.

To set breakpoint on message, OllyDbg creates specially prepared conditional logging breakpoint with explanation "<WinProc>". To see how it works, assume that we have button with handle 00001234 and request pause on all button messages. After breakpoint is set, open it as a conditional logging breakpoint. You will see the following options set:

```
Condition:      [ESP+4]==00001234 && [ESP+8] IN (0F0..0F7,135)
Explanation:    <WinProc>
Pause program:  On condition
```

At entry to window procedure, stack contains:

[ESP+00]	Return address
[ESP+04]	Window's handle
[ESP+08]	Message
[ESP+0C]	wParam
[ESP+10]	lParam

Now it's clear that first part of condition means: "Window's handle must be 00001234" and second: "Message is one of (BM_GETCHECK...BM_SETIMAGE, WM_CTLCOLORBTN)".

Important note: Window's handle is different each time window is created, so if you set breakpoint on actual window only, this breakpoint is valid only for the life of this window.

Message groups

Group	Messages in group
Any message	Any message
Creation and destruction	WM_CREATE, WM_DESTROY, WM_CLOSE, WM_QUERYENDSESSION, WM_QUIT,WM_ENDSESSION,

	WM_NCCREATE, WM_NCDESTROY, WM_INITDIALOG
Window activation	WM_ACTIVATE, WM_SETFOCUS, WM_KILLFOCUS,WM_ENABLE, WM_SHOWWINDOW, WM_CHILDACTIVATE, WM_QUERYNEWPALETTE
Window position and size	WM_MOVE, WM_SIZE, WM_QUERYOPEN, WM_SHOWWINDOW, WM_GETMINMAXINFO, WM_WINDOWPOSCHANGING, WM_WINDOWPOSCHANGED, WM_NCCALCSIZE, WM_SIZING, WM_MOVING, WM_ENTERSIZEMOVE, WM_EXITSIZEMOVE
Commands and notifications	WM_MEASUREITEM, WM_COMMNOTIFY, WM_NOTIFY,WM_NOTIFYFORMAT, WM_STYLECHANGING, WM_STYLECHANGED, WM_COMMAND, WM_SYSCOMMAND, WM_ENTERIDLE, WM_PARENTNOTIFY, WM_MDIRESTORE
System	WM_SYSCOLORCHANGE, WM_WININICHANGE, WM_DEVMODECHANGE, WM_ACTIVATEAPP, WM_FONTCHANGE, WM_TIMECHANGE, WM_COMPACTING, WM_POWER, WM_USERCHANGED, WM_DISPLAYCHANGE, WM_NCACTIVATE, WM_POWERBROADCAST, WM_DEVICECHANGE, WM_PALETTEISCHANGING, WM_PALETTECHANGED
Drawing	WM_SETREDRAW, WM_PAINT, WM_ERASEBKGND, WM_PAINTICON, WM_ICONERASEBKGND, WM_DRAWITEM, WM_NCPAINT, WM_QUERYNEWPALETTE, WM_PRINT, WM_PRINTCLIENT
Scrolling	WM_HSCROLL, WM_VSCROLL, WM_CTLCOLORSCROLLBAR
Icon	WM_QUERYOPEN, WM_QUERYDRAGICON, WM_GETICON, WM_SETICON
MDI	WM_MDICREATE, WM_MDIDESTROY, WM_MDIACTIVATE, WM_MDIRESTORE, WM_MDINEXT, WM_MDIMAXIMIZE, WM_MDITILE, WM_MDICASCADE, WM_MDIICONARRANGE, WM_MDIGETACTIVE, WM_MDISETMENU
Dialog	WM_CANCELMODE, WM_NEXTDLGCTL, WM_MEASUREITEM, WM_DELETEITEM, WM_GETDLGCODE, WM_CTLCOLORMSGBOX, WM_CTLCOLORDLG
Menu	WM_MEASUREITEM, WM_HELP, WM_CONTEXTMENU, WM_INITMENU, WM_INITMENUPOPUP, WM_MENUSELECT, WM_MENUCHAR, WM_ENTERMENULOOP, WM_EXITMENULOOP, WM_NEXTMENUWM_MDIREFRESHMENU
Text	WM_SETTEXT, WM_GETTEXT, WM_GETTEXTLENGTH, WM_SETFONT, WM_GETFONT
Mouse	WM_SETCURSOR, WM_MOUSEACTIVATE, WM_NCHITTEST, WM_NCMOUSEMOVE, WM_NCLBUTTONDOWN, WM_NCLBUTTONUP, WM_NCLBUTTONDBLCLK, WM_NCRBUTTONDOWN, WM_NCRBUTTONUP, WM_NCRBUTTONDBLCLK, WM_NCMBUTTONDOWN, WM_NCMBUTTONUP, WM_NCMBUTTONDBLCLK,

	WM_MOUSEMOVE, WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONDBLCLK, WM_RBUTTONDOWN, WM_RBUTTONUP, WM_RBUTTONDBLCLK, WM_MBUTTONDOWN, WM_MBUTTONUP, WM_MBUTTONDBLCLK, WM_MOUSEWHEEL, WM_XBUTTONDOWN, WM_XBUTTONUP, WM_XBUTTONDBLCLK, WM_CAPTURECHANGED
Keyboard	WM_VKEYTOITEM, WM_CHARTOITEM, WM_SETHOTKEY, WM_GETHOTKEY, WM_KEYDOWN, WM_KEYUP, WM_CHAR, WM_DEADCHAR, WM_SYSKEYDOWN, WM_SYSKEYUP, WM_SYSCHAR, WM_SYSDEADCHAR, WM_HOTKEY
Clipboard	WM_CUT, WM_COPY, WM_PASTE, WM_CLEAR, WM_UNDO, WM_RENDERFORMAT, WM_RENDERALLFORMATS, WM_DESTROYCLIPBOARD, WM_DRAWCLIPBOARD, WM_PAINTCLIPBOARD, WM_VSCROLLCLIPBOARD, WM_SIZECLIPBOARD, WM_ASKCBFORMATNAME, WM_CHANGECBCHAIN, WM_HSCROLLCLIPBOARD
Edit control	All EM_xxx messages
Static control	All STM_xxx messages
Button	All BM_xxx messages, WM_CTLCOLORBTN
Combo box	All CB_xxx messages, WM_COMPAREITEM
List box	All LB_xxx messages, WM_COMPAREITEM, WM_CTLCOLORLISTBOX
IME	All WM_IME_xxx messages
User-defined	All messages equal or above WM_USER

$#K# Heap list

Heap list is very similar to Memory map. It displays list of memory blocks allocated by debugged application on one of its heaps.

To walk heap, OllyDbg uses API functions *Heap32ListFirst*, *Heap32First* etc. defined in *kernel32.dll*. This API is not implemented on Windows NT and hangs OllyDbg on Windows 2000. I was unable to find any explanation to this behaviour, so heap list is currently supported only on Win95-based OSes.

$#K# Patches

Patches window (shortcut **Ctrl+P**) displays list of patches. From here, you can browse, apply or remove patches.

By definition, patch is a difference between global backup and actual code. If OS unloads some module, OllyDbg checks for differences in code section and saves them to .udd file. In the next session you can quickly restore them. Once in the list, patch is kept until overwritten or .udd file discarded.

Search for patches may be time-consuming, so Patches window doesn't update if you modify code. To freshen list of patches, select Actualize from the pop-up menu or press Patches button in toolbar.

$#KSearch for intermodular calls

Other types of search

This type of search looks for all calls to external modules (DLLs) and set breakpoint on API functions. Search includes all direct and indirect calls, calls to direct or indirect jumps, calls to Win95 thunks and their combinations. In many cases it successfully locates calls to functions loaded by *GetProcAddress* (but, of course, only after *GetProcAddress* was called).

You can sort found calls either by address of the CALL command or in alphabetical order. To find some specific function, simply start typing its name (without module name and preceding underscores).

$#K SEH window

SEH is an abbreviation of "Structured Exception Handling". If application makes something invalid, like division by 0 or call to pure virtual function, Windows raises an exception and looks for a handler that is ready to process it. In C++ code, you declare handlers using *try/catch* pairs.

Processing of exceptions must be code-related. If division by 0 happens in routine *Calculatemeanvelocity*, you may want to set mean velocity to 0. If this occurs in the rest of code, perharps the best solution is to write back modified files and terminate. To implement this behaviour, handlers are linked into the chain. Each elements contains two doubleword items:

```
addr+0  Pointer to next SEH record
addr+4  Address of SE handling function
```

Pointer=0xFFFFFFFF means end of the chain. Each thread has its own SEH chain, its start address is the first doubleword of the data block of the corresponding thread. To locate it, go to expression [FS:0]. On exception, Windows walks this chain and calls each handler asking whether it is able to process exception. If end of chain is reached, Windows calls default exception handler. In most cases, it displays message and terminates application.

SEH window displays the SEH chain for the thread selected in the CPU window. From this window, you may follow record or handler and set breakpoint on the entry of handling function. To avoid infinite recursion, total length of the chain is limited to 500 items.

$#KPlugins

Plugin is a DLL that resides in OllyDbg directory and adds functionality to OllyDbg. You can download free plugin development kit *plug110.zip* from the OllyDbg's homepage (http://home.t-online.de/home/Ollydbg).

Plugins can set breakpoints, add labels and comments, modify registers and memory. They can add menu items to main menu and many windows, like Disassembler, or Memory, intercept global and window-dependent shortcuts. They also can create own MDI windows. Plugins can write plugin-specific data to .udd files with module-dependent information and *ollydbg.ini* and access different data structures that describe debugged application. Plugin API includes more that 170 functions.

Many third-party plugins are available in Internet, for example, on OllyDbg forum http://ollydbg.win32asmcommunity.net, created and moderated by TBD.

To install plugin, copy DLL (and, if necessary, related files) to the plugin directory and restart OllyDbg. By default, this is the directory where main OllyDbg file *ollydbg.exe* resides.

Current distribution includes two "native" plugins: Bookmark and Command line. Their source is available in *plug110.zip*. These plugins are freeware, you can freely modify and redistribute them.

$#K Bookmark plugin

This plugin allows to set up to 10 code bookmarks using keyboard shortcuts or popup menus in Disassembler and then quickly return to one of the bookmarks using shortcuts, popup menu or Bookmark window. Bookmarks are program-specific. They are kept between sessions in .udd file.

To open window that lists all bookmarks, select Plugin|Booknmarks|Bookmarks from the main menu.

To manipulate bookmarks, use Bookmarks submenu in Disassembler window:

Bookmarks:

> **Insert bookmark 0** (Alt+Shift+0),
> **...**
> **Insert bookmark 9** (Alt+Shift+9) - set bookmark at the first selected line;
>
> **Delete bookmark 0,**
> **...**
> **Delete bookmark 9** - delete bookmark;
>
> **Go to bookmark 0** (Alt+0),
> **...**
> **Go to bookmark 9** (Alt+9) - follow bookmark.

$#K Command line plugin

Command line plugin is a simple command-line interface to OllyDbg. To open command line window, press **Alt+F1** or select Plugins|Command line|Command line from the main menu.

Command line plugin supports following commands:

Command	Description	Example
Expressions		
CALC expression	Calculates value of expression	CALC EAX/2+1
? expression	Ditto	
expression (first character is not a letter)	Ditto	2*2
WATCH expression	Add watch	WATCH +[460030+ESI]
W expression	Ditto	
Assignments		
SET reg=expression	Writes value of expression to 8-, 16- or 32-bit general register	SET AL=0 SET ESI=[DWORD EDI-10]
reg=expression	Ditto	AX=0FFFF
SET memory=expression	Writes value of expression to 8-, 16- or 32-bit memory	SET [410000]=80000001 SET [BYTE EAX+ESI*2]=0
Disassembler		
AT expression	Follow address in Disassembler	AT 410000
FOLLOW expression	Ditto	FOLLOW EAX
ORIG	Go to actual EIP	
*	Ditto	
Dump and stack		
D expression	Follow address in dump	D 460000
DUMP expression	Ditto	
DA [expression]	Dump in assembler format	
DB [expression]	Dump in hex byte format	
DC [expression]	Dump as ASCII text	DC EAX
DD [expression]	Dump as addresses (stack format)	
DU [expression]	Dump as UNICODE text	
DW [expression]	Dump in hex word format	
STK expression	Follow address in stack	
Assembling		
A expression [,command]	Assemble at address	A 410000, XOR EAX,EAX
Labels and comments		
L expression, label	Assign symbolic label to address	L EAX, loopstart
C expression, comment	Set comment at address	C EAX, Here loop starts
Breakpoint commands		
BP expression [,condition]	Set INT3 breakpoint at address	BP EAX+10 BP 410010, EAX==WM_CLOSE BP Kernel32.GetProcAddress
BPX label	Set breakpoint on each call to external 'label' within the current	BPX CreateFileA

	module	
BC expression	Delete breakpoint at address	BC 410010
MR expression1 [,expression2]	Set memory breakpoint on access to range	
MW expression1 [,expression2]	Set memory breakpoint on write to range	
MD	Remove memory breakpoint	
HR expression	Set 1-byte hardware breakpoint on access to address	
HW expression	Set 1-byte hardware breakpoint on write to address	
HE expression	Set hardware breakpoint on execute at address	
HD [expression]	Remove hardware breakpoint(s) at address	

Tracing commands

STOP	Pause execution
PAUSE	Ditto
RUN	Run program
G [expression]	Run till address
GE [expression]	Pass exception to handler and run till address
S	Step into
SI	Ditto
SO	Step over
T [expression]	Trace in till address
TI [expression]	Ditto
TO [expression]	Trace over till address
TC condition	Trace in till condition
TOC condition	Trace over till condition
TR	Execute till return
TU	Execute till user code

OllyDbg windows

LOG	View Log window
MOD	View Executable modules
MEM	View Memory window
CPU	View CPU window
CS	View Call Stack
BRK	View Breakpoints window
OPT	Edit options

Miscellaneous commands

EXIT	Close OllyDbg	
QUIT	Ditto	
OPEN [filename]	Open executable file for debugging	
CLOSE	Close debugged program	
RST	Restart current program	
HELP	Show this help	
HELP OllyDbg	Show OllyDbg help	
HELP APIfunction	Show help on API function	HELP CreateFile

Commands are not case-sensitive, parameters in brackets are optional. Expressions may include

constants, registers and memory references and support all standard arithmetical and boolean functions. By default, all constants are hexadecimal. To mark constant as decimal, follow it with decimal point. If you do not specify memory size, plugin assumes doubleword (32 bit).

Conditional logging breakpoint allows you to specify plugin commands that will be executed if break condition is met. To pass breakpoints to command line plugin, precede it with the point, for example:

```
.EAX=0
.RUN
```

Detailed description of examples:

- **CALC EAX/2+1** - calculates value of expression and displays it in hexadecimal format. Register belongs to the thread selected in CPU window;
- **2*2** - calculates value of expression (4) and displays it in hexadecimal format;
- **WATCH +[460030+ESI]** - adds watch that displays context of doubleword memory at address 0x460030+ESI in decimal and hexadecimal formats;
- **SET AL=0** - changes value of register AL in currently selected thread to 0;
- **SET ESI=[DWORD EDI-10]** - gets contents of doubleword memory at EDI-10 and writes it to ESI. Both ESI and EDI belong to the thread selected in CPU;
- **SET [410000]=80000001** - changes contents of doubleword memory at address 0x410000 to 0x80000001. Note that, unlike Disassembler or CPU Dump, command-line plugin does not create backup copy of modified memory block!
- **A 410000, XOR EAX,EAX** - changes command at 0x410000 to XOR EAX,EAX;
- **L EAX, loopstart** - assigns name "loopstart" to address EAX;
- **BP EAX+10** - sets unconditional breakpoint at address EAX+10;
- **BP 410010, EAX==WM_CLOSE** - sets conditional breakpoint at address 0x410010. Program breaks if register EAX is equal to symbolic constant WM_CLOSE;
- **BP Kernel32.GetProcAddress** - sets unconditional breakpoint at function GetProcAddress in module KERNEL32;
- **BPX CreateFileA** - set breakpoint on every call to external function CreateFileA from the module that is currently selected in Disassembler window.

$ OllyDbg
# OllyDbg
K Main index
$ What is OllyDbg?
# Whatis
K What is OllyDbg?
$ What's new in version 1.08
# C5OPFM
K What's new in version 1.08
$ Tips and tricks
# Tipsandtricks
K Tips and tricks

$ Apologies
# Apologies
K Apologies
$ Problems with localized versions
# localized
K Problems with localized versions
$ System requirements
# Systemrequirements
K System requirements
$ Installation
# Installation
K Installation
$ Support
# Support
K Support
$ Registration
# Registration
K Registration;What should I pay to use this shareware?
$ Legal part
# Legalpart
K Legal part
$ Your privacy and security
# privacy
K Your privacy and security
$ General principles
# Generalprinciples
K Debugging - general principles
$ How to start
# Howtostart
K Debugging - how to start debugging session
15J3UG5
IAP258
$ Debugging of stand-alone DLLs
# DebuggingofDLLs
K Debugging of stand-alone DLLs
$ Source code of LOADDLL.EXE
# LOADDLL
K Source code of LOADDLL.EXE
$ Analysis
# Analysis
K Analysis
WHYMIB
1YGJZ0I
$ Decoding hints
# Decodinghints

K Decoding hints
$ Object scanner
# Objectscanner
K Object scanner
$ Implib scanner
# ImplibS
K Implib scanner
$ Breakpoints
# Breakpoints
K Breakpoints
5D59HO
1WPGSY5
$ Shortcuts
# Shortcuts
K Shortcuts
$ Global shortcuts
# Globalshortcuts
K Shortcuts - global
$ CPU window
# CPUwindow
K CPU window
$ Context-sensitive API help
# Contexthelp
K Context-sensitive API help
$ Disassembler window
# Disassemblerwindow
K Disassembler window
$ Step-by-step execution and animation
# StepbyStep
K Step-by-step execution and animation
ENU0R8
$ Execute till return
# tillreturn
K Execute till return
$ Execute till user code
# tillusercode
K Execute till user code
$ Hit trace
# Hittrace
K Hit trace
$ Run trace
# Runtrace
K Run trace
9U7U6L
2PIW_RS

$ Self-extracting (SFX) files
# SFX
K Self-extracting (SFX) files
$ Information window
# Informationwindow
K Information window
$ Registers window
# Registerswindow
K Registers window
$ Disassembler shortcuts
# Disassemblershortcuts
K Shortcuts - disassembler
$ Command history
# Commandhistory
K Command history
$ Backup functions
# Backup
K Backup functions
$ Disassembler menu
# Disassemblermenu
K Disassembler window - menu
4AHHG83
S5LNN
72SBAD
3S7_90
KITPK1
23E70TP
XHXHZ1
3NVX0B
IBKDN9
$ Appearance menu
# Appearance
K Appearance menu
$ Dump
# Dump
K Dump
$ Dump menu
# Dumpmenu
K Dump window - menu
13TJRDR
F2N1.9
$ Stack window
# Stackwindow
K Stack window
$ Executable modules window

modules
K Executable modules window
$ Memory map
# Memorymap
K Memory map
$ Watches and inspectors
# Watches
K Watches and inspectors
$ Threads
# Threads
K Threads
$ Evaluation of expressions
# expressions
K Evaluation of expressions
$ Options
# Options
K Options
$ General options
# Optionsgeneral
K Options - general
$ Defaults
# Optionsdefaults
K Options - defaults
$ Dialog options
# Optionsdialogs
K Options - dialogs
$ Directory options
# Optionsdirectories
K Options - directories
$ Font options
# Optionsfonts
K Options - fonts
$ Colour options
# Optionscolours
K Options - colours
$ Code highlighting options
# 2M54_LE
K Options - code highlighting
$ Security options
# Optionssecurity
K Options - security
ORLBPN
$ Debug options
# optionsDebug
K Options - debug

$ Event options
# Optionsevents
K Options - events
$ Exceptions options
# Optionsexceptions
K Options - exceptions
1IN6JVC
$ Trace options
# optionsTrace
K Options - trace
3V3OB
2HRPZH8
$ SFX options
# optionsSFX
K Options - SFX
$ String options
# Optionsstrings
K Options - strings
$ Address options
# Optionsaddress
K Options - addresses
D2YUY4
$ Command options
# Optionscommands
K Options - commands
1VJPZWQ
$ Disassembler options
# Optionsdisassembler
K Options - disassembler
$ CPU options
# optionsCPU
K Options - CPU
$ Register options
# Optionsregisters
K Options - registers
VCPY4K
$ Analysis options - part 1
# Optionsanalysis1
K Options - analysis 1
$ Analysis options - part 2
# Optionsanalysis2
K Options - analysis 2
$ Analysis options - part 3
# Optionsanalysis3
K Options - analysis 3

$ Stack options
# optionsStack
K Options - stack
$ Just-in-time debugging
# YTQGIP
K Just-in-time debugging
$ Register tracing - an example
# TS90EK
K Register tracing - an example
$ Assembler
# Assembler
K Assembler
1YQGQBJ
$ Disassembler
# Disassembler
K Disassembler
$ Labels
# Labels
K Labels
3LVWKGE
HLXTUB
$ User comments
# comments
K User comments
$ References
# References
K References
$ Known bugs and problems
# Knownbugs
K Known bugs and problems
$ How the Analyzer works - an example
# example
K Analysis - an example
$ Custom function descriptions
# Customfunctions
K Custom function descriptions
LEDAFI
42CL_VF
SF4DRD
$ Search
# Search
K Search
$ Names window
# Names
K Names window

$ SSE support
# SSE
K SSE support
$ Binary search
# Binarysearch
K Search for binary strings
$ Search for a sequence of commands
# 6P9_MQ
K Search for a sequence of commands
$ Search for modified commands or data
# 1MOJG5Z
K Search for modified commands or data
$ Profile
# Profile
K Profile
$ Call stack
# Callstack
K Call stack
$ Call tree
# Calltree
K Call tree
$ Windows
# Windows
K Windows
$ Heap list
# Heaplist
K Heap list
$ Patch manager
# PatcheS
K Patch manager
$ Search for intermodular calls
# intermodularcalls
K Search for intermodular calls
$ SEH window
# SEHwindow
K SEH window
$ Plugins
# Plugins
K Plugins
$ Bookmark plugin
# Bookmark
K Bookmark plugin
$ Command line plugin
# Commandline
K Command line plugin