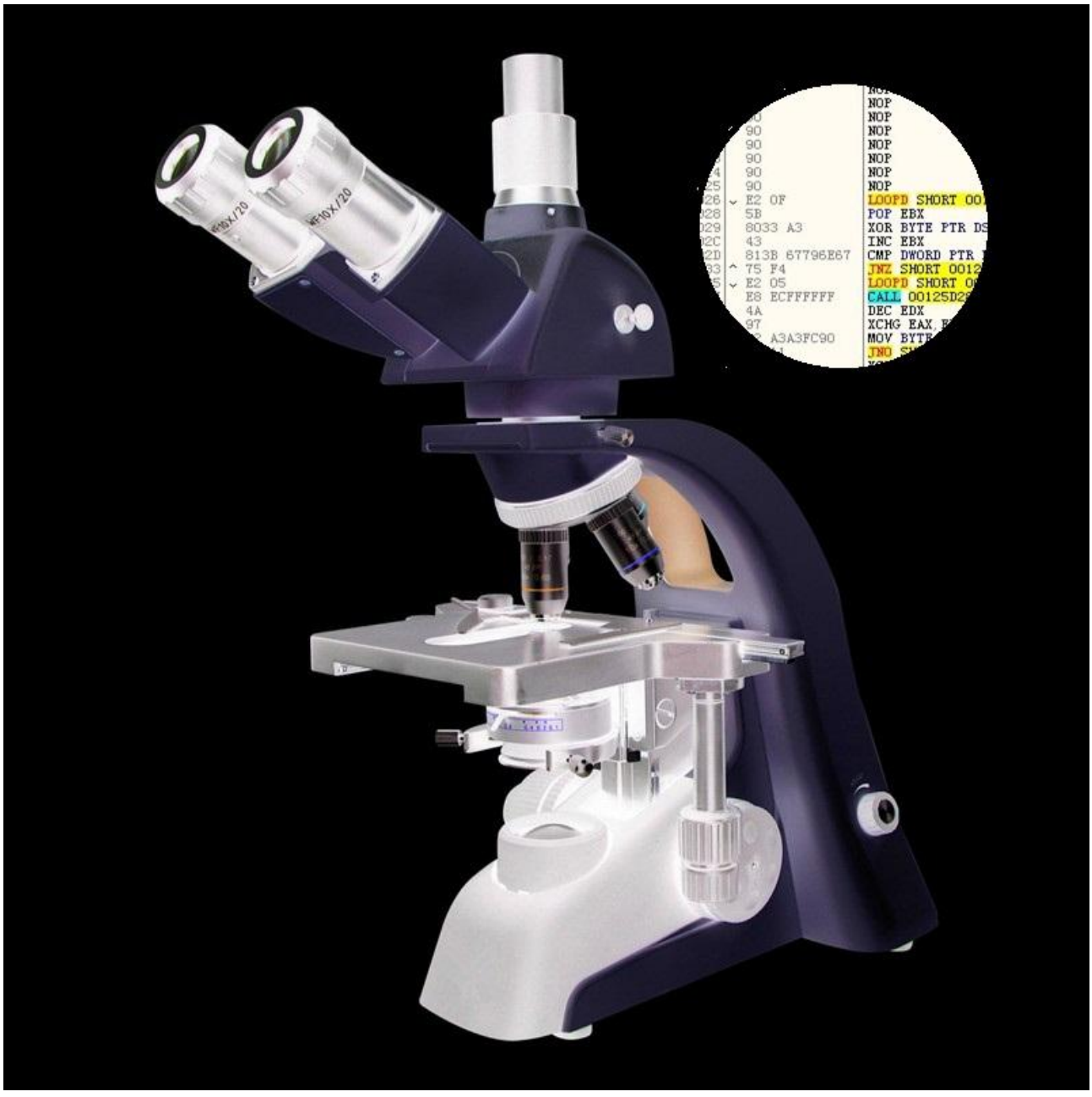
Ollydbg 简明帮助手册

中文版



Version 2.0.1

Oleh Yuschuk　著

安天安全研究与应急分析小组（安天 CERT）献译

前言

Ollydbg 是一款结合 IDA 和 SoftICE 功能的调试工具，因其简易的操作和强大的功能，目前已成为安全研究领域使用最广泛的调试解密工具。目前 Ollydbg 软件已发布 2.0x 版本，但内附的帮助手册尚无中文翻译版发布。基于合作分享，互惠互利的理念，同时为更多新版 Ollydbg 的使用者提供便利的条件，安天 CERT 对帮助手册展开翻译工作，并形成当前的译文版本。现将中文版帮助手册与安全研究领域的从业者分享，希望能够对新版本 Ollydbg 的使用者有所帮助。译文存在的错误和不足之处，欢迎大家批评指正。

——安天安全研究与应急分析小组（安天 CERT）

免 责 声 明	本译文译者为安天实验室工程师，本文系出自个人兴趣在业余时间所译，本文原文来自互联网的公共方式，译者力图忠于所获得之电子版本进行翻译，但受翻译水平和技术水平所限，不能完全保证译文完全与原文含义一致，同时对所获得原文是否存在臆造、或者是否与其原始版本一致未进行可靠性验证和评价。 本译文对应原文所有观点亦不受本译文中任何打字、排版、印刷或翻译错误的影响。译者与安天实验室不对译文及原文中包含或引用的信息的真实性、准确性、可靠性、或完整性提供任何明示或暗示的保证。译者与安天实验室亦对原文和译文的任何内容不承担任何责任。翻译本文的行为不代表译者和安天实验室对原文立场持有任何立场和态度。 译者与安天实验室均与原作者与原始发布者积极联系未果，亦未获得相关的版权授权，鉴于译者及安天实验室出于学习参考之目的翻译本文，而无出版、发售译文等任何商业利益意图，因此亦不对任何可能因此导致的版权问题承担责任。 本文为安天内部参考文献，主要用于安天实验室内部进行外语和技术学习使用，亦向中国大陆境内的网络安全领域的研究人士进行有限分享。望尊重译者的劳动和意愿，不得以任何方式修改本译文。译者和安天实验室并未授权任何人士和第三方二次分享本译文，因此第三方对本译文的全部或者部分所做的分享、传播、报道、张贴行为，及所带来的后果与译者和安天实验室无关。本译文亦不得用于任何商业目的，基于上述问题产生的法律责任，译者与安天实验室一律不予承担。
注 ：	如有原作者联系方式或信息请与我们联系。

 © squashed bug 2.0

OllyDbg © 2000-2013 Oleh Yuschuk， 版权所有

OllyDbg 中使用的全部品牌名称和产品名称、附带的文件或帮助的均为商标、注册商标或其各自所有者的商标名称。在提及来源的前提下，您可以由此帮助文件自由摘录。

千里之行，始于足下

——中国格言

目录

介绍

1.10 与 2.xx 版本的区别

实际上，32 位调试器 OllyDbg 的第二版是经过推倒重来地设计的。因此，它比上一版本更快、更强也更稳定。好吧，至少以后会是这样，这是因为有些 1.10 版本的实用功能在 2.01 版尚无法使用。 2.01 版本具有一些 1.10 版本不曾拥有的新功能。包括：

- 全面支持 SSE 和 AVX 指令集。SSE 寄存器可直接访问，无需经过代码注入。
- 在调试器上下文执行命令时，可以跟踪执行速度——包括执行条件和记录——最高支持每秒执行 1，000，000 条命令。
- 内存断点数量不受限制。
- 内存和硬件可设置条件断点。
- 更为可靠，分析时不需要借助跟踪。
- 分析器可识别未知函数的参数个数，有时还支持参数含义的识别。
- 支持脱离（Detaching）被调试进程。
- 支持子进程的调试。
- 提供在 TLS 回调时暂停的选项。
- 提供未处理异常过滤器，可放行未处理异常。
- 内建整数和 FPU 命令帮助。
- 可定制快捷键。
- 用户界面支持多语言。

OllyDbg 2.01 概述

OllyDbg 2.01 是一款 32 位汇编级分析调试器，界面直观，在没有源代码或解决编译级别问题的时候，尤其适用。

所需环境：虽然本工具主要在 Windows XP 环境下开发和测试，但可在各种 32 位版本 Windows 系统运行，包括：NT、2000、XP、2003 服务器版、Vista、Windows 7 等等。不再支持早期基于 DOS 的版本（如 95、98、ME） 。这就是说，如果你安装了 Microsoft Layer for UNICODE[1]，你也许可以启动 OllyDbg ，甚至可以设置断点，但我并不保证能够稳定操作。2.01 版尚无法工作于 64 位 Windows。 为使调试工作显得舒适，你至少也要有 1-GHz 主频的 CPU。OllyDbg 很占内存。如果你打开全部功能调试大型应用程序，它会轻易用掉 200~300 兆内存来备份和分析数据。

支持指令集。OllyDbg 2.01 支持全部现有 80x86 兼容 CPU：MMX、3DNow!、Athlon 扩展、包括 SSSE3 和 SSE4 在内的 SSE 指令，以及 AVX 指令。

可配置性。具有超过 120 个控制 OllyDbg 行为和表现的选项（哦，不对！目前还没有那么多！）。

数据格式。Dump 窗口可按全部常见格式显示数据：十六进制、ASCII、多字节、UNICODE、16/32 位有符号/无符号/十六进制整数、32/64/80 位浮点数、地址、反汇编（MASM、IDEAL、HLA 或 AT&T 格式）。同时可以解析和标注多种 Windows 特有结构，包括 PE 头、PEB、线程数据块等。你还可以 Dump 系统内存（仅限 XP）、文件及原始磁盘。

使用帮助。OllyDbg 2.01 提供了全部 80x86 整数、浮点指令的内建帮助。如果你拥有 Windows API 帮助文件（win32.hlp，因为版权问题未予提供），可以加进来，从而在系统 API 调用时获得即时帮助。

程序启动。你可在命令行指定调试文件、由菜单选择、向 OllyDbg 窗口拖拽文件、重新启动最后调试程序，或者附加到运行中的应用程序。OllyDbg 支持实时调试（just-in- time debugging）和调试子进程。你可以脱离调试进程，该进程可继续执行。无需安装！

代码高亮。反汇编器可以对不同类型的指令（跳转、条件跳转、压栈出栈、调用、返回、特

1 简称 **MSLU**，是一种为创建 **Unicode** 软件而提供给开发者的软件库。——译者注

权指令和非法指令）和不同运算（常规、FPU/SSE 或段/系统寄存器，栈上内存或其它内存运算，常数） 加亮。你可以创建自定义高亮方案。

线程调试。OllyDbg 可以调试多线程应用程序。你可以在线程之间切换，挂起、恢复和杀死线程或者改变其优先级。线程窗口会显示各线程的错误（根据 GetLastError 调用的返回结果）。

分析功能。分析器是 OllyDbg 至关重要的部分。它可以识别过程、循环、switch、表、GUID、常量和嵌入代码的字符串、复杂的结构、API 函数调用、函数参数个数、导入节，等等。对于未知函数，它不仅会尝试确定其栈上的参数个数，甚至还会尝试判断这些参数的意义。分析功能可以让二进制代码变得更为清晰、易于调试，也减少了误判和崩溃的概率。该功能不依赖于具体编译器，对任何 PE 程序均可胜任。

全面支持 UNICODE。 所有 ASCII 字符串可用的功能，均适用于 UNICODE，反之亦然。OllyDbg 能够识别 UTF-8 字符串。

命名（Names）。OllyDbg 认识许多常量的符号名（目前是 10800 个），比如窗口消息、错误码或位字段，并可在已知函数调用中对其解码。

已知函数。OllyDbg 认识超过 2300 个常用 Windows API 函数，并能对其参数解码。你还可以自行添加函数描述。你也可以在已知或猜测的函数设置日志断点（logging breakpoint ），将参数规范记录下来。

调用（Call）。OllyDbg 可以向后跟踪（backtrace ）栈上的嵌套调用，哪怕调试信息已经失效或者调用过程使用了非标准的序言（prologs ）和结语（epilogs）[2]。

堆栈（Stack）。OllyDbg 在堆栈窗口使用了启发式识别返回地址和栈结构的方法。如果程序暂停在已知函数，堆栈窗口会对已知或猜测的函数参数解码。堆栈功能还可以跟踪和显示结构化异常（SE）的处理函数链。如果集成的堆栈遍历功能不好用，你也可以切换到使用 Dbghelp.dll 的方式。

搜索功能。相当多！可以搜索单个指令（精确或模糊）或指令序列，搜索常量，搜索二进制或文本字符串（不必连续），搜索所有涉及地址的指令，搜索常量或地址范围，搜索所有到指定位置的跳转，搜索全部文本字符串的引用，搜索全部交叉调用，在整个已分配内存中使用掩码搜索二进制序列，搜索整型或浮点型数字等等。如果找到了多个位置，你可以快速地

2 即由编译器在函数开始和结束时生成的代码。——译者注

在它们之间切换。

断点功能。 OllyDbg 支持所有常见断点类型：软件断点（INT3 或其它几条指令）、内存断点和硬件断点。你可以指定放行次数及设置暂停条件。也可以有条件地将断点数据记录到日志中。软件断点和内存断点的个数不受限制：在命中跟踪的极端情况下，OllyDbg 可以设置成百上千的 INT3 断点。如果 CPU 足够快，OllyDbg 处理断点的速度可达每秒 2 万到 3 万次。

监视器。监视器会在程序每次暂停的时候计算表达式。你可以使用任意复杂的寄存器、常量、地址表达式、布尔及代数操作

执行。你可以单步执行程序、进入子例程或让程序立即执行。你可以让程序执行到下一次返回，执行到指定位置，在深层嵌套的系统 API 中通过反向跟踪回到用户代码。在程序运行时，你可以完全地控制它。比如说，你可以查看内存、设置断点，甚至在运行中修改代码。无论什么时候，你都能够暂停或重新执行所调试程序。

命中跟踪（Hit trace）。命中跟踪功能会显示到目前为止执行过哪些指令或过程，该功能供你测试代码的全部分支。命中跟踪由当前位置开始执行，在各个尚未跟踪到的分支设置软断点。这些断点会在指令被执行（命中）后移除。

运行跟踪（Run trace）。运行跟踪功能会以单步模式执行程序，且约定都传输到大的缓冲区里。运行跟踪速度很快：在快速指令仿真功能被启用的情况下，OllyDbg 的跟踪速度可达每秒 1 百万条指令！ 运行跟踪功能会记录寄存器（不含 SSE/AVX）、标志位、已访问内存的内容、线程错误，以及——针对具有自修改能力的代码——修改前指令。你可以指定停止运行跟踪的条件，比如地址范围、表达式或指令。你还能将运行跟踪的结果保存到文件里，然后比较两次运行的结果。运行跟踪功能支持向后跟踪，以及对千百万条指令的执行的历史详细分析。

概要（Profiling）。概要功能（profiler）可计算出运行跟踪缓冲区里面列出的一些指令的执行次数。借助概要功能，你会了解哪部分代码用掉了最多的执行时间。

补丁（Patching）。内置汇编器会自动选用尽可能短的代码。二进制编辑器支持同时以 ASCII 、UNICODE 和十六进制形式显示数据。以往的复制粘贴功能仍然有效。自动备份功能支持取消所做的修改。你可以直接向可执行文件复制修改后的内容。OllyDbg 甚至可以对其适当修正。

UDD。OllyDbg 将全部程序及关联模块的信息都保存在独立的文件中，在重新加载模块

时，再恢复这些信息。这些信息包括标签、注释、断点、监视、分析数据和条件等等。

UDL。你可以将编译器所支持的标准库转换为 UDL 库，分析器会据此识别代码中的库函数。

插件（Plugins）。你可以通过编写自己的插件（或由互联网下载插件），扩展 OllyDbg 2.01 的功能。**定制化。**你可以指定自定义的字体、颜色和高亮方案。

文字到语音转换（Text-to-speech）。该功能适用于视觉障碍用户：激活文字到语音转换选项后，OllyDbg 可以朗读当前选中的内容。

还有更多！本列表远远谈不上完整，还有更多特色功能，这些功能让 OllyDbg 2.01 成为一款令人满意的调试器。

免注册

OllyDbg 2.01 版权属于 Oleh Yuschuk， 版权所有 2000-2013。这是一款闭源的免费软件。作为用户，，无论是出于私人用途还是商业用途（取决于许可协议），你都可以免费使用 OllyDbg。无需注册。我曾经在最初的版本里采用了注册机制，这只是为了了解我的程序有多么流行。结果远远超出预期。注册单令我不堪重负，显著地降低了我的生产力。因此：OllyDbg 2.01 是免注册的。

如果您是将 OllyDbg 应用于教学目的的教授或老师，我乐于与您取得联系（Ollydbg@t-online.de），如果您有何令我的产品更易于学习或更能满足教学目的的建议，尤其欢迎。

法律部分

商标信息。OllyDbg 中使用的全部品牌名称和产品名称、附带的文件或帮助的均为商标、注册商标或其各自所有者的商标名称。使用它们仅出于识别目的。

许可协议。本许可协议（以下简称为“协议”）附带于 OllyDbg 2.01 版本及相关文件（以下简称为“软件”）。使用本软件即说明你已同意本协议全部条款及规定之约束。

本软件版权属于 Oleh Yuschuk（以下简称为“作者”），版权所有 2000-2013 Oleh Yuschuk。你被允许免费使用本软件。你可在任意数量的存储设备（如硬盘、内存记忆棒等）上安装本软件，并可将本软件备份任意数量。

本软件依原样提供给用户，并不做出明确或暗示的担保，这些担保包括但不限于软件用于特定目的的适用性。在任何情况下，作者都不会承担在正常使用、误用或无法使用本软件时造成的，任何特殊的、附带的、间接的、直接的或任何其它的利润损失或保留损失，即使作者已被告知这种损失的可能性。作者尤其不会对使用附加到本软件的第三方插件造成的危害负责。

除非且仅限于扩展本软件功能之外，你不可修改、反编译、反汇编本软件，或对本软件采用逆向工作方法，此类活动受相应法律明确限制。你不可单独发布或使用本软件的部分内容。在满足以下条件的情况下，你可以制作或发布本软件的复制品：a) 复制品包含原始发布时的全部文件，且这些文件未被修改； b) 若你随同本软件发布了任何其它文件，这些文件必须按此形式明确标明，而且其使用条款不能超出本协议的约束条件；c) 你不能收取费用

（除非为传输介质，如 CD），即使你在发布时包含了附加文件。本协议仅适用于当前 OllyDbg 2.01 版本。对于其它版本，受与此类似的独立许可协议保护。

公平使用。多数软件厂商会明确禁止你尝试反汇编、反编译、使用逆向工程方法或修改其程序。该限制了会涉及到你的应用程序所使用的所有第三方动态库，包括系统库。如果你有任何疑问，请联系版权所有者。这里所说的“公平使用”，可能会导致误解。你可能想 You may want to discuss whether it applies in your case with competent lawyer. 请勿将 OllyDbg 用于非法用途！

你的隐私和安全。在我正确无误上传文档（包括 OllyDbg.exe 及其支持文件）到互联网的前提下（以下简称“原生的 OllyDbg”），以下声明适用于版本 1.00 到 2.01。这些文件不包括任何第三方插件。

对于原生的 OllyDbg，我担保以下内容:

- 永远不会试图窥探所调试程序之外的进程，或有类似网络客户端/服务端（行为，或出于任何意图发送任何数据到任何计算机（除非是用户指定的远程文件），或存在类似任何类型木马的行为，只有一种情况会例外： 如果你允许 dbghelp.dll 访问微软符号服务器（Microsoft Symbol Server），该 DLL 可能会与微软产生数据交换。对应的选项默认是关闭的，你需要明确将其激活。
- 除非有明确请求，否则不会读写系统注册表。该请求会受以下两个注册表键值影响：

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

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

- 除非你明确请求，否则不会创建、重写或修改任何系统目录文件。
- 除非有明确请求，否则不会修改任何计算机上的任何可执行文件或 DLL，包括 OllyDbg 本身。
- 仅在你明确请求后，才会记录你的调试活动（不包括保存于 ollydbg.ini 的文件历史和记录调试信息的.udd 文件）。甚至我还可以担保，在没有得到你的允许的前提下，OllyDbg 只会在其所在目录及选项（Options）对话框指定的目录内创建或修改文件。
- （最后一条，但 by no means least）不会包含显式或隐式的强迫你注册本程序的“唠

叨”（nag）界面，也不含有任何依赖注册或过期失效的功能限制。

小心病毒。尽管我已使用多种反病毒软件检查过原生的文件，但不能排除 OllyDbg 发行版会受到病毒感染，或者会有木马伪装成 OllyDbg 及其所需文件。对于你的计算机受到任何由病毒导致的损害，或者受到任何由于木马附着在本软件的部分文件、全部文件以及由于第三者对于 OllyDbg 的修改而造成的损害，本人概不负责。

安装

OllyDbg 无需安装。简单地建立一个新的文件夹，再将 Odbg201.zip 解压至该文件夹即可。如有必要，可拖拽 ollydbg.exe 的连接到桌面，为其建立一个快捷方式。在 Windows 7 平台下，需要在该快捷方式的属性中，为其激活“以管理员身份运行”（"Run as administrator"）。

如果你是一个坚持要在 Windows NT 4.0 系统下运行 OllyDbg 的顽固派（没有问题！），你会需要用到 psapi.dll。该库文件并未不包含在本软件中。2.01 版本不打算支持在基于 DOS 的 Windows 版本运行，但是，通过安装 MIcrosoft Layer for UNICODE，是可以做到这一点的。一些很老很老的版本，比如 Windows 95 并不包括 VirtualQueryEx 和 VirtualProtectEx 两个 API 函数。这些函数对于调试功能非常重要。如果 OllyDbg 提示缺少这类函数，正常的调试功能将无法进行。请升级你的操作系统吧。

总之，Windows 95、98 和 ME 已经不再被支持了。不妨使用 OllyDbg 1.10， 它是免费的，而且兼容于这些类系统。

支持

http://www.ollydbg.de 网站提供有限支持。在该网站可以下载错误修正及新版本程序。如果你遇到问题，可发送邮件到 Ollydbg@t-online.de。一般我会在一周内回复。如果你没有明确禁止，我保留摘录你邮件内容并发布到我的网站的权力。

全部源代码均可提供，不过需要付钱。这些源代码是完全自主实现的（“clean-room”），并未使用任何第三方代码。你既可以购买完整的全部源代码，也可以只购买其中的一部分，比如反汇编模块，汇编模块或分析模块。想了解更多，可以发邮件给我。顺便提一句，反汇编模块（Disassembler） 2.01 版本已经按 GPL v3 协议发布，可以我的主页下载。

各版本 OllyDbg 的.udd 文件格式描述均可以免费索取。

本帮助所使用的设置选项

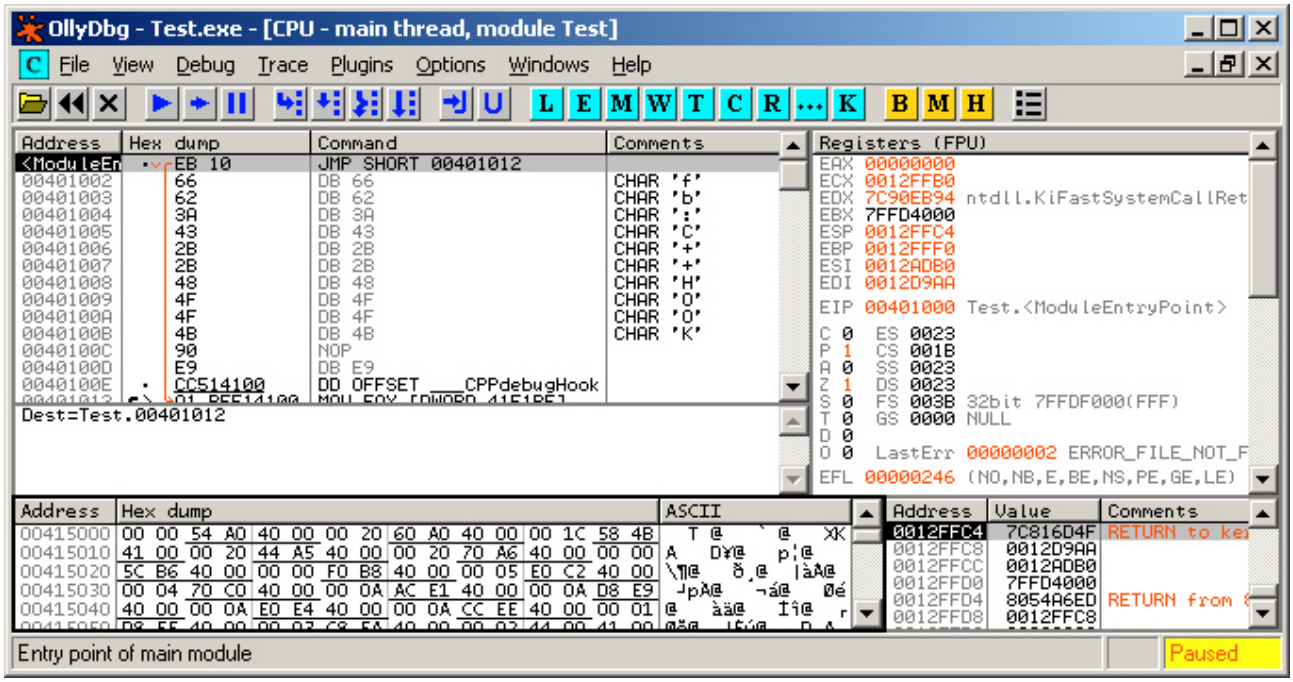
一些选项的设置会影响到 OllyDbg 的外观及功能。本帮助文件假定这些选项均使用默认设置。这些选项了也包含快捷键设置。多数 OllyDbg 的快捷键都是可以重新设置的。如果你的快捷键设置与此不同，可通过菜单操作，或恢复为默认设置：选项（Options ）->编辑快捷键（ Edit shortcuts...）->恢复默认（Restore defaults）。

第一步 启动一个应用程序

熟悉一个新的程序最好的办法就是练习。假设你已经简要的熟悉 80x86 处理器的体系结构和指令集，也了解进程线程和模块的概念，并且知道可移植的可执行文件由头和节组成。

此处有惊喜!在 OllyDbg 分布档案有一个小的测试应用程序命名为- -Test.exe。

启动 OllyDbg，并加载 test.exe。你可以从资源管理器中拖拽这个测试应用程序，或按 F3，或选择文件|从主菜单中打开。OllyDbg 会打开应用，分析它的代码并在主模块的入口点暂停。（主模块是 Test.exe 本身）：



你将会看到很多取决于 OllyDbg 行为的控制选项，类似于上面的图片。OllyDbg 是一个老式的没有浮华修饰的 MID 应用程序。其外观已被性能优化。默认的小字体和窄边框可以使你在显示屏上看到最大化的信息量。当然，你随时可以调整颜色和字体使他们适合你的眼睛。

大多数的 OllyDbg 的屏幕上面充满了 CPU 窗口。这是你将要花费大部时间调试应用程序的窗口。他由五个窗格组成：

反汇编	寄存器
信息	
Dump	堆栈

反汇编指令列表存储在指定的内存中。对于每一条指令，它的信息包括在内存中的十六进制地址，它的标签，二进制指令代码，反汇编后的指令名称以及指令的注解。

如果应用程序中途停止，地址栏中当前执行点（寄存器指针 EIP 所指向处）会被标记为黑色。

大多数的注解是由反汇编人员以及分析人员编写的。它们可以帮助你理解代码的含义。你当然也可以添加你自己的注解。

如果你是一个初学者，你可以选择一条指令，然后按 Ctrl+F1[3]来获取帮助。（目前仅支持整数和 FPU 指令）。

信息窗口显示关于第一条被选指令的附加信息。它可能包括操作数的内容和它的解码，已知跳转的列表和调用指令。对于一条可以被识别的条件指令来说，会包含跳转是否实现，如果实现，那么它的源代码行，循环变量等信息。

寄存器窗口显示所选线程的寄存器内容。字母 C，P，A 等等代表单独的标志（进位标志、奇偶标志、辅助进位标志，.....）。这些标志是标志寄存器 EFL 的一部分。Last err 模块不是一个寄存器，它显示的是 Windows API 函数存储错误代码在内存中的内容。例如，如果调用 CreateFile（”abc.def，…，OPEN_EXISTING…）无法找到 abc.def 文件，那么 last error 就会被置为 0x00000246（ERROR_FILE_NOT_FOUND）。

如果寄存器或位的内容自从上一次暂停改变了或者被用户修改了，那么就会被高亮显示。

FPU 寄存器是由 FPU，MMX 和 3D!NOW 指令集共同使用的。要想改变他们的显示，使用鼠标右键弹出的菜单或者在窗口顶部点击相应的按钮。

堆栈窗口显示与所选线程关联的堆栈。在地址栏，ESP 所指向的地址会被标黑。OllyDbg 会尝试识别堆栈框架并从内容的左边开始用括号来标记。它可能还会高亮显示可能是返回值的 doublewords。但是要注意，它们也可能是以前函数调用的遗留。在堆栈窗口会显示一个堆栈的精简视图。

Dump 显示的是主模块的数据区域。你可以选择任意位置和不同的数据演示。Dump 是一个独立的线程。

底部的状态栏会显示调试信息和当前的调试状态。右下角的红色“暂停”表示应用程序被 OllyDbg 挂起。另一些可能的状态是“加载”（Loading），“运行”（Running），“跟踪”（Tracing），“终止”（Terminated）等。

[3] Ctrl+F1 与 WIN32API 帮助有冲突——译者注

第一课 断点

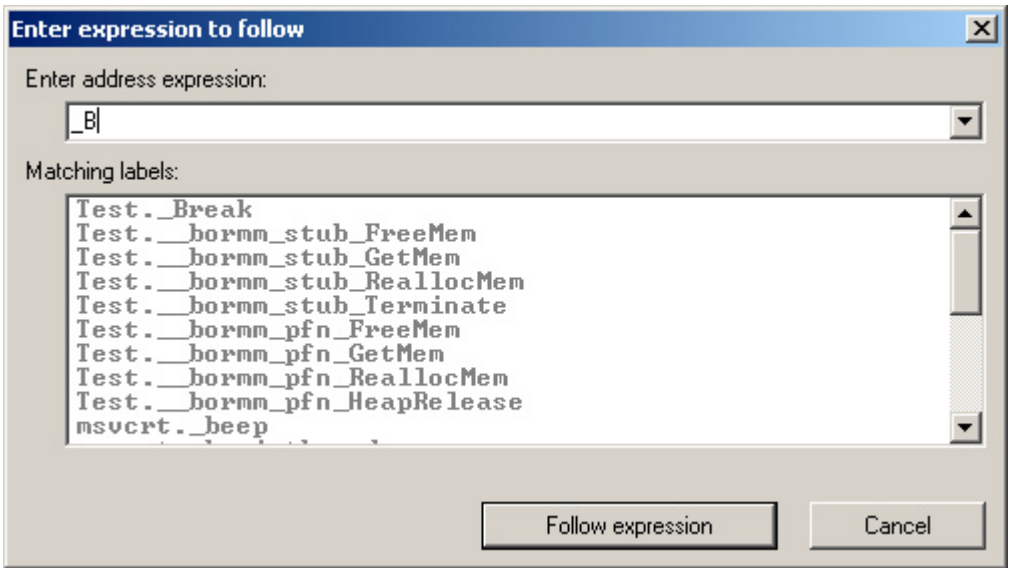
简单来说，断点是为了使程序能够在执行或者访问某一地址时请求暂停而设置的。Ollydbg 也支持条件断点（当某些条件（如 EAX==1）满足时即暂停执行）和断点的记录功能（在不停止程序的情况下将一些数据记录到日志中）。

Ollydbg 支持三种断点：软件断点、硬件断点和内存断点。在设置软件断点时，调试器会利用一些在执行时会报出异常或者中断的命令替换断点位置命令的第一个字节。一般来说，经常使用 INT3 （二进制代码 0xCC）作为替换指令，某些特权指令，如 HLT 或者 CLI 等，也可以作为替换指令。

如上文所述，将 Test.exe 载入 Ollydbg 时，状态将改变为“暂停”（如果没有，打开“选项|开始”选项，将第一个暂停位置更改为**程序主模块的入口点**，并重新载入样本文件）。在主菜单中选择“调试|运行”，或者直接按 F9 键或点击直角三角形的按钮后，状态将变为“正在运行”，并且弹出程序的主界面。

点击按钮**“Read [_Break]”**您会看到 **Read [_Break] = 0x90（NOP）**这样的信息。当您按下按钮时，测试程序会读取字节到标志 _Break 的位置处，并显示内容。

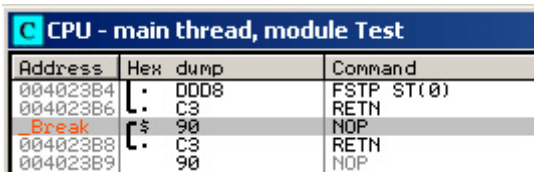
通过点击画面内部某处，我们将焦点转向反汇编窗口，您有没有注意到窗口周围黑色的窄的轮廓？现在，按下 **CTRL+G**（或者右键点击屏幕选择 **GOTO|Expression** （鼠标指向转到，后指向表达式））然后输入“_Break”。



从出现的窗口中，您可以改变反汇编窗口中显示的内存地址。您可以使用直接的地址，如 401023（十六进制地址 0x00401023），象征性的标签，像_Break 或类似 ECX + EDX * 4 的表达式的形式。请注意，所有的常量都是 16 进制。100 指的是$0x100_{16} = 256_{10}$，100.

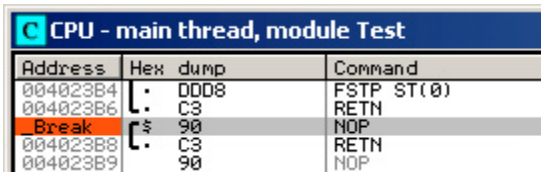
指的是100_{10}（100.0 指的是浮点数 100.0，在地址表达式中是不允许出现的。

在键入数据时，在列表的底部会显示以所输入文本为子串的标签列表。我们要寻找的标签是 Test 模块中的_Break 字段，因此它的全称是 Test_Break。点击一次，并且点击**接下来的标签**（因为某些原因，双击操作无效）。



在反汇编重新定位到_Break 程序的位置时，需要注意，Test.exe 是一直驻留的，且地址可能与屏幕上显示的不同。程序包含了两条指令，一条是 RETN，另一条是紧随其后的 NOP，它们在这里不做任何事情。_Break 标签指向的是十六进制代码为 0x90 的 NOP 指令，这就是处在调试状态中的应用程序给出的信息。

NOP 指令的这一行会一直处在被选中的状态么？现在，按 **F2** 键，就是设置断点的快捷方式。红色背景表示所设的断点为无条件软件断点。如下图所示，Ollydbg 利用 INT3 指令取代了 004023B7[4]地址出的指令。



再选择一次"**Read [_Break]**"，该地址的信息就会变成"**[BYTE _Break] = 0xCC (INT3)**"。因此，这个进程的内存信息在物理上发生了变化。但是，反汇编工具会仍然显示修改之前的命令么？是的，因为在内存信息显示在窗口之前，反汇编工具就将断点信息替代为原来的数据了。但是，正如我们看到的那样，软件断点在 Ollydbg 工具的窗口上是可见的。例如，一些病毒样本在没有设置断点时仅仅检查关键的系统功能，并不去做坏事，但是如果设置了软件断点，它可能产生我们意想不到的恶意行为。所以，**Ollydbg 工具只允许我们在命令的第一个字节设置断点！**

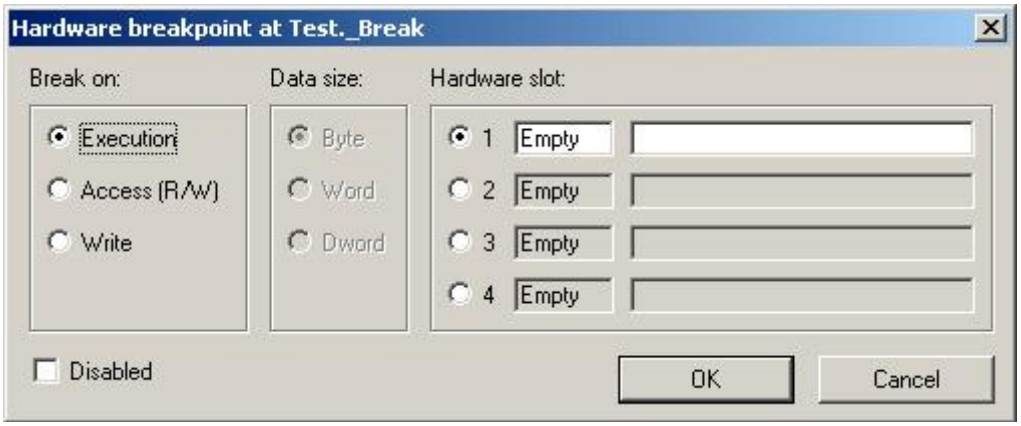
断点的设置和使用都很灵活，点击"**Call _Break**"，Ollydbg 弹出断点信息。此时调试过的程度会变成“暂停”状态，同时会出现"**Breakpoint at Test._Break**"的消息。这时发生了什么？INT3 命令使得程序处于中断状态，Windows 系统暂停了程序的运行，并将中断请求发送至调试器。这时你就可以做读寄存器内容、分析内存或者打补丁这些你想做的事了。

按下 **F9** 或者点击“运行”按钮，程序就能继续执行。此时设置断点的那一行指令断

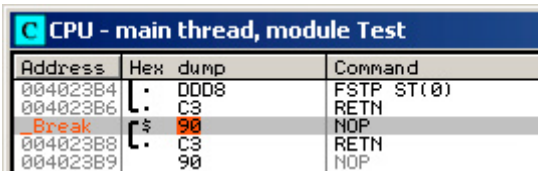
[4] 原版帮助手册中此处地址为 004023AB，此处应为作者错误——译者注

点仍然存在，选中该行，按下 F2，即可移除断点。

现在，单击鼠标右键，并在弹出的菜单中指向“断点”，并在附加菜单中选择“硬件执行”，如下图所示，Ollydbg 会向你确认这个断点触发的条件：命令执行、内存访问、内存写入。因为此时内存地址会被当做代码一样识别。Ollydbg 工具默认推荐的是命令执行选项。

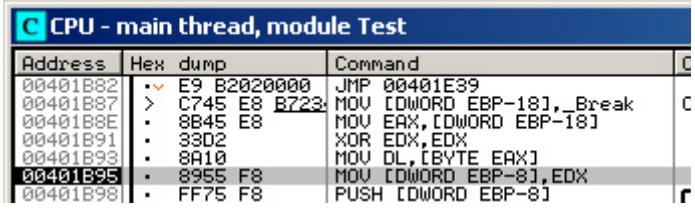


点击确定，就会弹出新的对话框。内存 Dump 窗口的红色的背景表示硬件或者内存断点。



点击"**Read [_Break]**"，二进制代码是 NOP，硬件断点对程序是不可见的。（也不一定是这样，程序会读取线程的内容，也可以分析调试寄存器）。按下"**Call_Break**"就会命中断点。

运行程序，调用硬件断点的对话框，将断点类型改为（**R\W**）模式。按下"**Call_Break**"，没有什么异常发生，按下"**Read [_Break]**"。程序就停止运行，但是程序没有命中到_Break 标签的地址。



请注意状态栏的消息。

数据的硬件断点是在数据被访问之后触发的。检查 EAX 寄存器，在上面的示例中，它的内容是 004023B7。这就是_Break 标签的地址。这个地址是从 00401B93 位置的命令 **MOV DL，[BYTE EAX]**.读取来的。但是硬件断点记录的是下一条命令的地址。

硬件断点是非常方便实用的。如果你在数据中错误地下了硬件断点，什么异常都不会发生。如果内存地址显示是一条数据，那么硬件断点是不会触发的 。但是硬件断点是非常稀少的，只有四个。关于硬件断点的属性，有一定的局限性。所以该学习内存断点了。

现在，我们跳过硬件断点。在运行时，EIP 指向下一个命令的地址，而你却不知道怎么跳转回来？你可以在窗口上调用 GoTo，也可以点击寄存器上的 EIP[5]寄存器，并从菜单中选择 **Follow in Disassembler** 选项。你也可以点出反汇编菜单，选择 **GoTo**（转向），并在弹出的附加菜单中选择 **minus**（上一个）或者直接在小键盘区按下减号键即可（当然，如果你有小键盘区的话）。反汇编器和 Dump（转储）窗口会保留不同的历史地址，让我们能够从两个方向去运行程序。

移除硬件断点之后，选择 **Breakpoint | Memory...**（断点|内存断点）然后点击执行，并运行程序。一些奇怪的事情发生了：状态栏在闪烁。闪烁的频率就好像每秒 6000 次一样。内存断点使用的是 80x86 的内存保护机制，如果不可用的内存在执行中被访问，那么该地址每次被命令命中，硬件都会产生一次异常。如果异常是误报，Ollydbg 会继续执行，但是异常数量会显示在状态栏上。**异常处理需要很长时间，所以内存断点会有些慢**，但是内存断点的数量是不受限制的。

按下"**Read [_Break]**".，正如我们预想的那样，内存断点在程序中不可见，按下"**Call[_Break]**".，Ollydbg 才会报出程序中内存断点的位置。

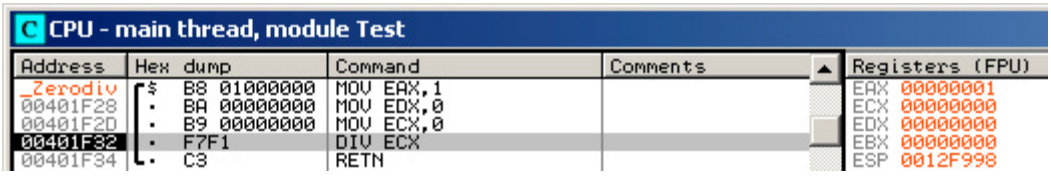
[5] 原版帮助手册中此处为 EAX，改为 EIP（作者的问题？）——译者注

第二课 给代码打补丁

运行 Test.exe(无 OllyDbg)，然后按下标记为“**0:0**”的按钮。程序将调用整数 1 整除整数 0 的例程，正如预期的那样，程序崩溃：

我们要找到无效的指令并将其从代码中删除。加载 Text.exe 到 OllyDdg 中并运行（**F9**）。再次按下“**0:0**”按钮。执行将会暂停并在底部状态栏显示消息“**Integer division by zero - Shift+Run/Step to pass exception to the program**”。

查看 CPU 反汇编窗体：



EIP 指向指令 **DIV ECX**。如果你不知道这是什么指令，按 Shift+F1（或从菜单中选择“**帮助**”）：

查看寄存器，一眼可知寄存器 ECX 的值为 00000000，即被零除。我们是在子程序中用下面的代码标记为_Zerodiv：

MOV EAX，00000001

MOV EDX，00000000

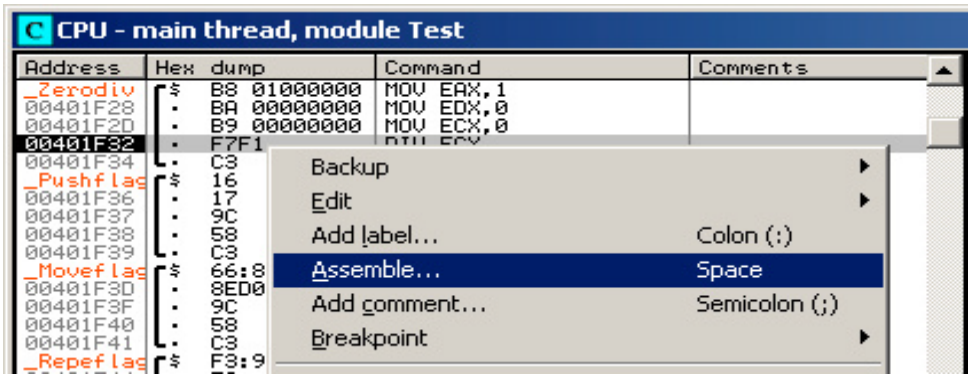
MOV ECX，00000000

DIV ECX

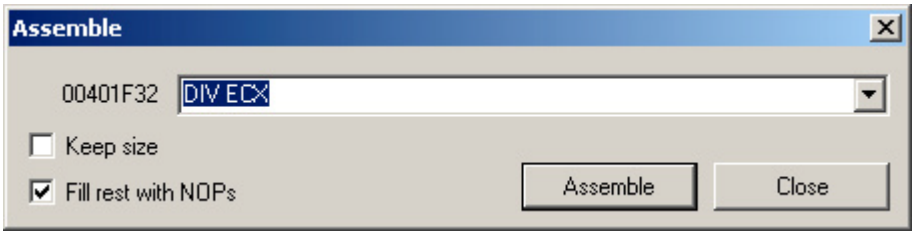
RET

现在我们有几个选择。我们可以对整个程序进行修改，更换 MOV EAX，00000001 与 RET。或者我们将 ECX 的值置 1。或者我们可以用 NOP 指令替换 DIV ECX。我们选择第三种方法。

点击 DIV ECX。当这个指令被选中时，右键单击它，然后在菜单中选择 **Assemble**（**汇编**）[6]：



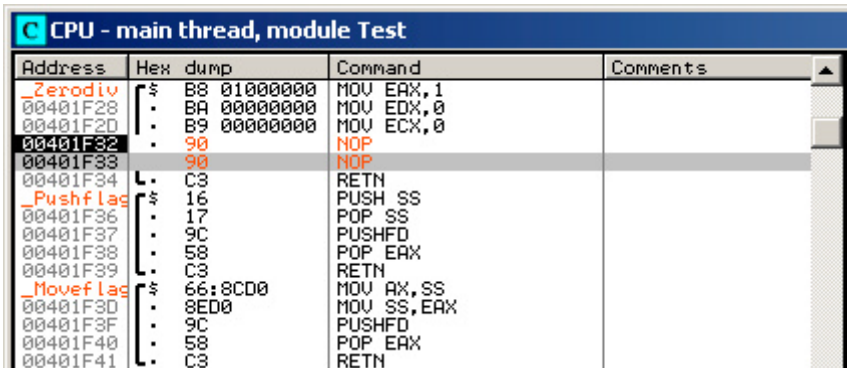
汇编窗口将出现：



注意两个复选框。如果要检查大小，你将只能修改选定的内存（在我们的例子中，DIV ECX 的长度是 2 字节，只有地址 00401F32 和 00401F33）。这样可以防止重要的代码被重写。

新的指令不一定要和原来的长度相同。假设你想删除跳转到目的地址是 8 字节的二进制代码 EB F4，你可以用 NOP(90)代替他，不过 NOP 只有 1 个字节长。你将得到序列 90 F4 来拆解 NOP。如果你尝试执行此序列，程序将崩溃，报告特权指令。不受控制的指令残存可能是危险的。因此，我建议总是激活填充 NOP 指令。如果新的指令不完全合适，其余的将被自动填充无害的 NOP 指令。

回到本章，选择 **Fill rest with nop**（**填充 NOP 指令**）、NOP 类型并按下**汇编**（或键盘**回车**键）。窗口要求你输入下一条指令，但我们不需要输入。关闭对话框，你将会看到：



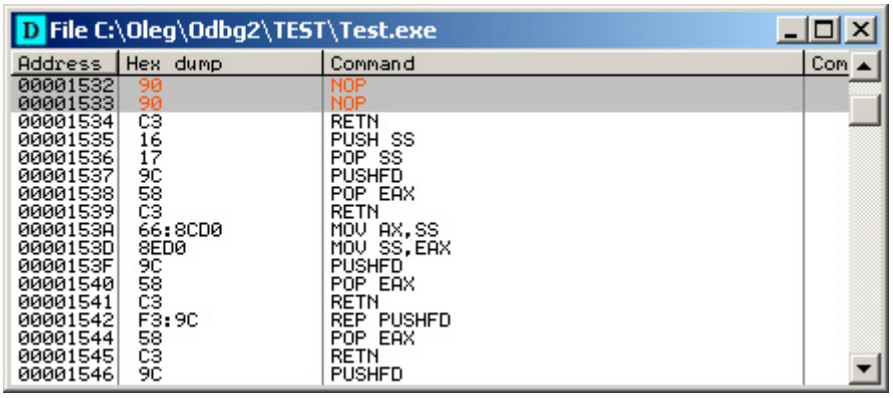
DIV ECX 指令是由两个 NOP 指令序列替代。请注意修改的命令已设置为高亮。为了与

[6] 此处的快捷键为空格键——译者注

原命令保持联系，OllyDbg 创建了旧编码的备份。此操作消耗内存（备份和编码段一样大）但非常实用。你可以查看旧的编码，撤销更改或搜索修正。（如果代码在调试时自动修改，那么备份将会起到作用）。

还要注意的是括号表示该过程_Zerodiv 现在是损坏的。Ollydbg 移除了修改后代码的分析。

运行程序并再次按下“**0:0**”，什么事情都没有发生，这说明我们的修补程序是正确的，但当应用程序终止时更改将丢失。我们需要将它们复制到可执行文件中。这是一个复杂的任务，但 OllyDbg 会应付它。选择修改过的编码（两个 NOP 指令），调用上下文菜单并选择**编辑|复制到可执行文件**。OllyDbg 将创建新的 Dump 窗口，读取可执行文件，查找对应的地址，并将修改的部分复制到 Dump 文件中：



你可以添加其他补丁，或者直接通过 CPU 反汇编程序。当你准备好后，点击右键文件 Dump 并选择**保存文件**…你将被要求确认此操作，只需按 YES。另一个对话框会出现，要求你选择文件名，我们不想改变原来的 Test.exe 名（如果测试应用程序人在运行，此操作将不被允许）。选择不同的名字，比如说，test1.exe 并**保存**。

运行 test1.exe 按“**0:0**”。没有例外，我们已经成功地修补程序。

第三课 运行跟踪

当程序执行跳转到错误的位置时，我们会很难发现这个无效跳转的位置。

而这里有一个叫做运行跟踪的功能。当你启动运行跟踪功能时，OllDbg 调试应用程序时，会一次一个指令的一步步执行。

运行跟踪是非常非常缓慢的。现代的 CPU 一秒钟可以执行数十亿条指令。但当运行跟踪时，执行速度会被限制到大约每秒 3 万条指令。如果快速指令仿真功能开启

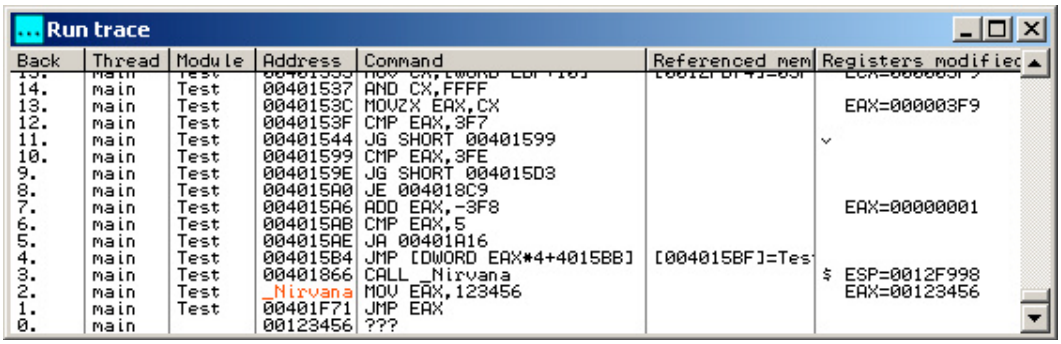
（**Options|Debugging|Allow fast command emulation**），OllyDbg 通常每秒能跟踪 3 万到 6 万条指令。这个速度对简单的 GUI 应用程序来说已经足够了，比如 Test.exe

运行跟踪的另一个缺点是它目前只能跟踪一个线程。如果是由两个交叉线程所引起的错误，运行跟踪功能是不能发现的。但是 Text.exe 只有一个线程。

现在开始本节的主要内容。在 OllyDbg 中打开 Test.exe，然后按下标有“**JMP 123456**”的按钮。注意！OllyDbg 会报告“访问冲突：读取[00123456]时”，但是 CPU 反汇编是空的。这是因为内存地址[00123456]是空的。

继续尝试。设置运行跟踪选项（**选项|运行跟踪**）。对我们来说，只有选项“**禁止进入系统 Dll**”是有用的。它一定是不可选的，因为跳转被 Windows 回调函数执行。重启应用程序（**调试|重启**）并运行它（或者点击）。调试模式是“on the fly”时是不允许更改的。暂停程序并开启运行跟踪。注意这时状态改变为“跟踪”并且状态栏显示类似“120672 events per second”的字样。运行跟踪会触发许多调试事件，通常都是执行的命令。切换到应用程序窗口，然后按“**JMP 123456**”。

为了避免程序崩溃，这时你应该注意错误提示“OllyDbg 不能跳过 0x123456 地址的指令（可能是无效的）。内存不可读。”。取消它并且打开运行跟踪协议（视图|运行跟踪）或按钮：



最后一个指令是跳转到地址 00123456。最后一个 JMP EAX 的地址是 00401F71。正如从最后一列看到的那样，这时 EAX 的值是 00123456。我们已经找到问题的根源了。

Text.exe

Text.exe 能够让你学习 OllyDbg 的扩展功能。在本节中，我们已经运用了这些功能。本手册中的大多数代码示例都是基于此应用程序。

你可以从我的网站 www.ollydbg.de 上下载 Test.exe 的源代码。我会不断的增加新的功能和更新现有的功能。因此地址可能会有所改变，也会有新的按键被添加。可以使用导出标签来找到代码。

下面是每个按键的简要说明。括号中的斜体文字是相关的标签名称，如下：

Start thread(*_Thread*)：以 100 毫秒的间隔（相当不准确）启动打开窗口和计数的新的线程。如果关闭窗口，线程终止；

Suspend last：暂停上一次创建的线程。你可以在 OllyDbg 的线程窗口恢复它；

Hide last：试图从调试器中隐藏上一次创建的线程（NtSelnformationThread（），code 0x11）。这个功能依赖于操作系统。如果不支持隐藏，就会看到错误信息；

FatalExit（）：调用 FatalExit（0）；

New process：启动 Test.exe 的新实例。如果选项调试子进程被检查，OllyDbg 会启动新的实例并将其附加到新创建的进程；

New suspended：与上面相同，但新的进程最初处于暂停状态（不运行）；

Load ws2_32：加载库 ws2_32.dll。为什么加载 ws2_32 呢？因为它适用于所有的 Windows 版本并且会加载另一个库 ws2help.dll；

Unload ws2_32：从内存中卸载 ws2_32.dll 库；

Current Dir-显示 GetCurrentDirectorry()函数所获取的当前目录。

Sleep(5000)-执行 Sleep(5000)即暂停 5 秒。

Set filter-通过调用 SetUnhandledExceptionFilter()来设置自定义的过滤器，用以处理不能未处理的异常情况。注意，通常不允许调试器向这个过滤器传递异常。OllyDbg 使用一些小技巧来搞定 OS 使之能够调用这个过滤器。这个行为由选项 **Pass unprocessed exceptions to Unhandles Exception Filter** 控制。

Set VEH-给向量化异常处理器链(VEH)添加句柄。为了保护应用程序免遭病毒的破坏，Windows 对这此链使用会话唯一(session-unique)标志技术。因此只有当进程是由 OllyDbg 创建，以及进程不是由 OllyDbg 依附得到时，才可以使 VEH 正常工作。

Read [00000000] (_Accessviolation) -使 Test.exe 读取内存地址为 00000000 处的内容。这通常会导致访问违法。但是你可以通过 **ZwAlloc(0)**来在地址 0 处分配内存块!

0:0 (_Zerodiv) - 用 0 除以 0，导致整数除以 0。

INT3 (_Int3) - 执行 INT3 指令。

INT ff (_Intff)- 执行 INT oFFh 指令。

JMP 123456 (_Nirvana) - 跳转到未分配的内存页 JMP 0123456h。

Stack overflow - 启用无限递归调用，使栈溢出。

1.0:0.0 (_Fzerodiv) -用浮点数 0.0 除以浮点数 1.0，由浮点数除法中用 0.0 作除数引发。注意由于浮点协同处理器的逻辑性，这个错误将会在下一个 FPU 命令中报出。为了找到无效命令，你必须在寄存器面板中查看 Last cmnd 参数。

Set Trap (_Settrap) - 设置寄存器 EFL 中位 T(单步陷阱)，由单步异常引起。

Hard BP in thread -在上一个创建的线程中设置硬件调试断点 0，由单步异常引起(所有的调试异常都被定位为相同的中断，windows 并没有区分它们)。

POP SS/PUSHF (_Pushflags) -执行命令 PUSH SS; POP SS;PUSHFD 并显示栈顶的内容。在 OllyDbg1.10 跟踪这个序列的时候，会压入标志 T。

MOV SS/PUSHF (_Moveflags) - 执行 MOV AX，SS;MOV SS，AX;PUSHFD 命令，和前一个按钮效果相类似。

REPE PUSHF (_Repeflags) - 执行未公开命令 REPE:PUSHFD。在 OllyDbg1.10 跟踪这个命令的时候，压入标志 T。

INT 2D (_Int2d) - 执行 INT 02Dh 命令。

String A - 执行 OutputDebugStringA("Debug string (ASCII)")。

String W - 执行 OutputDebugStringW(L"Debug string (UNICODE)")。

Read [_Break] - 读取并显示地址_Break 处的字节。这个地址是一个规模较小从未执行过的例程入口。

通常，在你按下按钮的时候，测试例程会报告[BYTE Break] = 0x90 (NOP)。尝试在_Break 地址处设置软件断点并再次按下按钮。现在读到的消息是:[BYTE Break] = 0xCC (INT3)，说明 NOP 被 1 个字节的 INT3 中断替换了。

Call _Break -调用上面已经描述过的_Break()函数。除了设置一个断点外，该函数上面也没有做。

ZwAlloc(0) - 在基地址 0x00000000 处分配 1 页(4096 字节)。是的，这也是可以做到哒!

Set vars (_ppi and _rect in data section) - 内部计数器加 1 并将其值赋予变量**ppi and rect[2][3].right。这些变量声明如下:

int **ppi;

RECT rect[4][4];

在分析中，也有几个地方进行了高亮:

内嵌循环-5 个短小的内嵌循环:

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

内嵌调用-函数调用中以函数调用作为参数使用:

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

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

汇编器和反汇编器

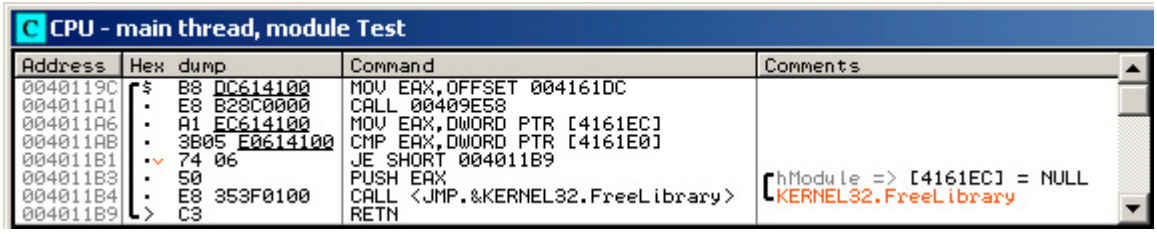
信息概述

OllyDbg 2.01 支持全部 80x86 指令、FPU、MMX、3DNow!、SSE、SSE2、SSE3、SSSE3、SSE4 和 AVX 扩展。请注意，以下特性与差异是相对于 Intel 标准而言：

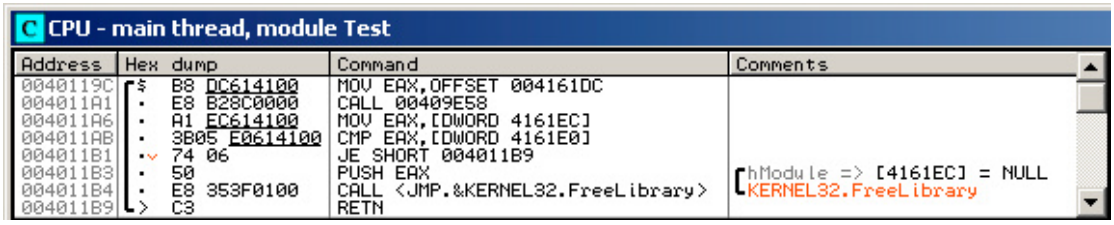
- 支持 REP RET(AMD 转移预测错误修正)；
- 支持多字节 NOP(如 NOP [EAX])，然而汇编器总是试图选择最短的形式，因此他可能很难设置所需的 NOP 长度；
- FWAIT 通常从 FPU 指令中分离，汇编器不添加 FWAIT，例如：FINIT 实际上翻译为 FNINIT 等；
- 汇编解释二进制 FPU 指令无操作数形式(如 FPU)，但反汇编总是用两个操作数形式(FADDP ST(1)，ST);
- LFENCE:只以 E8 的形式接受(0F AE E8)；
- FMENCE:只以 F10 的形式接受(0F AE F0);
- SFENCE:只以 F8 的形式接受(0F AE F8);
- PINSRW:16 位解码寄存器(通常是低 16 位)；
- PEXTRW:按英特尔标准内存操作数是不被允许的；
- 有些 FPU、MMX 和 SSE 指令只接受寄存器或内存中的 ModRM 字节。如果对方没有定义，反汇编器报告它作为未知指令。整数指令如 LES，在无效的操作数的情况下报告。
- SSE4 指令使用寄存器 XMM0 作为第三个可用的操作数在 2-和 3-操作数之间的格式，但反汇编中，只显示完整的 3-操作数形式；
- 汇编接受有明确 AL/AX/EAX 的 CBW、CWD、CDQ、CWDE 作为操作数。反汇编只显示隐含的无操作数形式；
- 16 位 AVX 浮点数是十六进制短号码解码；

反汇编模式

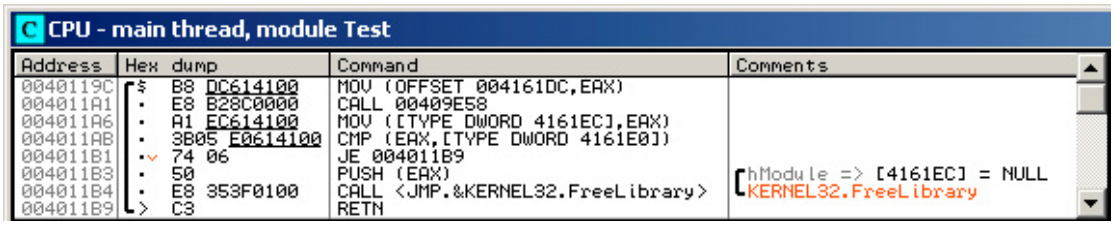
OllyDbg 支持四种不同的解码方式：MASM，Ideal，HLA 和 AT&T。它们是有由 **Code | Disassembling syntax（代码|反汇编语法）**选项控制。MASM 实际上是标准的 80x86 的汇编程序：



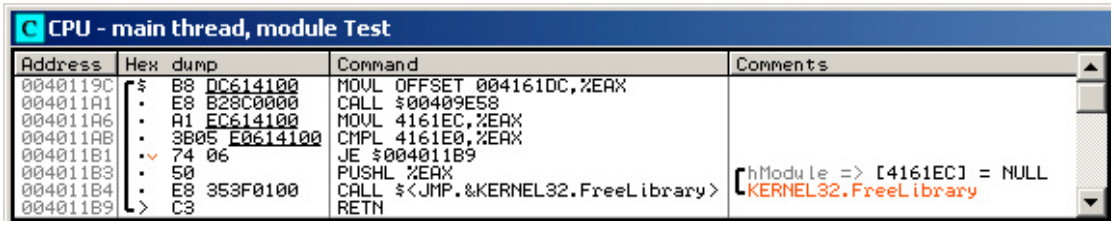
理想模式，由 Borland 推出，与 MASM 非常相似，但内存地址的解码方式不同：



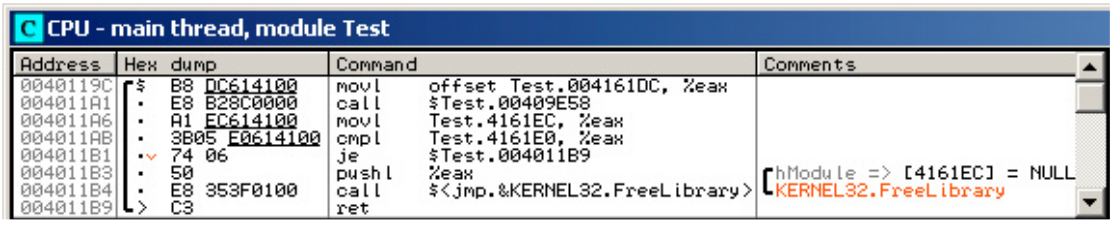
高级汇编语言，由 Randall Hyde 创建，它的语法是第一个操作数是一个源操作数并且操作数在括号中。HLA 是一个公共领域的软件，你可以将文档、教程、源代码一起下载，网址如下：http://webster.cs.ucr.edu。HLA 语法示例：



AT&T 语法：在 Linux 程序员中非常流行：



反汇编是可配置的。下面是 AT&T 模式的另一种设计：



符号名称的识别

在 c++中，你可以用不同数量或类型的参数声明相同名称的函数。但接口必须能够区分它们。为了确保这一点，编译器增加了参数和函数返回值类型的描述。这个过程叫做名称修饰或者 name mangling。例如，void Setdlgscroll(HWND hparent， int id， int pos)

在 Visual Studio 中会被编码为？Setdlgscroll@@YAXPAUHWND_@@HH@Z；

在 Borland C++ Builder 中会被编码为@Setdlgscroll$qp6HWND_ii。

在 Test.exe 中的 Floatingcall（）和 Floatingargs（）函数的 mangled name 为@Floatingargs$qfdg 和@Floatingcall$qv。

恢复原来的名字叫做 demangling。OllyDeg 能够对 GNU，微软，Borland 的编译器创建的名称进行重命名。这个过程由选项 **Code|Demangle symbolic names** 控制。不像 OllyDeg v1.xx 版本那样，v2 版本可以在两个版本中随时转换。

尽管可以提取参数的数量和类型，OllyDbg 也只有恢复函数名称的功能。如果选择了重命名，可能一些内存位置会被命名成相同的。

尽管可以提取出参数的数量和类型，但是 Ollydbg 只存储函数的名称。如果解码开始进行，那么可能会发生几个内存地址命名相同的事情。

条件命令

看一下这两个命令：

83F8 00 CMP EAX，0　　09C0 OR EAX，EAX

74 50 JE 00123456　　74 51 JE 00123456

他们的功能是一样的，如果 EAX 的值为 0，就执行跳转命令。在第一种情况下，助记符 JE 描述了跳转的触发条件：EAX=0.那么第二个例子中的 JE 代表的是什么？两个 EAX 相等么？当然不是，JE 的触发条件是 OR EAX EAX 的表达式为零。在这种情况下，助记符 JZ 会更好一些。但是问题是，JE 等同于 JZ，两个指令的操作码都是 74（当偏移量超过 128 字节时，操作码会变成 0F84）。Ollydbg 是如何判断哪一种解析方式更好的？答案是分析。当 Ollydbg 分析代码时，它会注意那些设置条件标志的指令，并以标志为依据选择助记符。该功能在助记符中这个选项中有所体现。猜一下代替条件指令的形式，以及影响 JE、JZ、

JNE、JNZ、JAE、JC 以及 JB、JNC 等命令解码的因素。

汇编语法

汇编器会自动识别 MASM，Ideal 和 HLA 语法。AT&T 语法是不支持的。无论选择哪种反汇编方式你都可以键入 MASM、Ideal 或 HLA 格式的命令。HLA 指令是围绕着数字和括号进行识别的。下面是几条 HLA 命令

MOV [EAX]，ESI

MOV [DWORD EAX]，ESI

MOV DWORD PTR [EAX]，ESI

MOV (ESI，[EAX])（请注意 HLA 操作数相反的顺序）内存操作数要求使用方括号，MOV EAX，_Zerodiv 指令的行为是将_Zerodiv 标签的地址移动到 EAX 寄存器中。MOV EAX，[_ Zerodiv]的行为是将_ Zerodiv 地址出的双字长度的内容移动到 EAX 寄存器中。MOV _Zerodiv，EAX 指令是无效的。

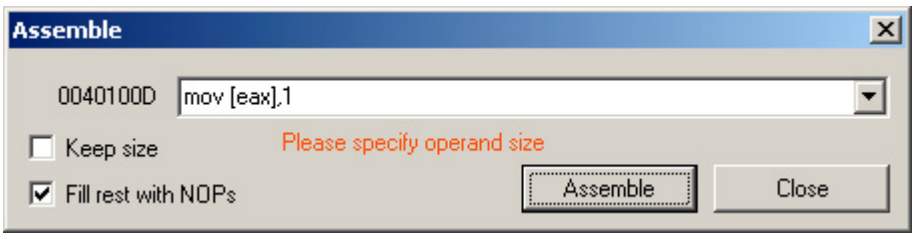
如果一个操作数起源于另一个操作数，或由其他操作数派生，那么它无需指定大小。例如，汇编器已经知道 ESI 寄存器的大小是双字（4byte），MOV [EAX]，ESI 的操作数也是四个字节。但是在 MOV [EAX]，1 指令的情况下，三种操作数长度都可能出现。

MOV [DWORD EAX]，1

MOV [WORD EAX]，1

MOV [BYTE EAX]，1

此时汇编器将发出警告。



你可能已经熟知，常量可以直接用于命令操作数:

MOV EAX，WM_PAINT 转换为 MOV EAX，0F

操作数中的表达式仅限于加法，减法，常量及标签的或运算。这三种操作具有相同的优先级，执行顺序从左到右。忽略 32 位溢出，不允许有括号:

MOV EAX，WS_CHILD|WS_VISIBLE 转换为 MOV EAX，50000000

OllyDbg 中所有的常量默认都是 16 进制，如果你想声明一个十进制常量，按照下面来做

(译者注前面的指令末尾没有句点):

MOV EAX，1000 等价于 MOV EAX，0x1000

MOV EAX，1000. 转换为 MOV EAX，3E8

16 进制常量以一个字母(A-F)开头，但是符号标签比十六进制数字具有更高的优先级。假设你在地址 0x00401017 处定义了一个标签 DEF，那么：

MOV EAX，ABC 转换为 MOV EAX，0ABC

MOV EAX，DEF 转换为 MOV EAX，401017

为避免混淆，在 16 进制常量前缀以 0 或者 0x: MOV EAX，0DEF。

对于这条规则，有一些例外:参数索引以及局部变量都是十进制。比如，ARG.10 是第十个调用参数的地址，其偏移是 10_{10} 4=40_{10}=0x28。注意到 ARG 和索引之间以小数点间隔，你可以以此来记忆这条规则。

序数也是以十进制来表示，COMCTL32。#332 就表示是第 332_{10} 次导出。

也支持 16 位地址，但是汇编器总是假设 32 位的代码段添加已 0x67 为前缀的地址大小的代码段。如果地址不包含 16 位寄存器，可以使用关键字 SMALL 来强制使用 16 位地址模式:

MOV ECX，[DWORD FS:0] 生成 32 位地址

MOV ECX，[SMALL DWORD FS:0] 生成 16 位地址(命令长度减少了 1 字节)

未公开的 80x86 命令

OllyDbg 识别几个未公开的 80x86 命令:

命令	操作码	注释
INT1(ICEBP)	F1	1 字节断点
SAL	D0 /6，D2 /6，C0 /6	算术移位，等同于 D0 /4
SALC	D6	设置 AL 进位标志
TEST	F6 /1	逻辑测试，等同于 F6 /0
REPNE LODAS， REPNE MOVS，...	F2:AD， F2:A5，...	字符串操作，REPNE 的 解释方式等同于 REP
FFREEP	DF /0	释放浮点寄存器
UD1	0F B9	保留的未定义指令

		(Intentionally undefined instruction)

反汇编器支持上述提到的所有命令.汇编器不会产生非标准的 SAL 和 TEST 指令。在必要情况下，使用二进制编辑来创建二进制代码。

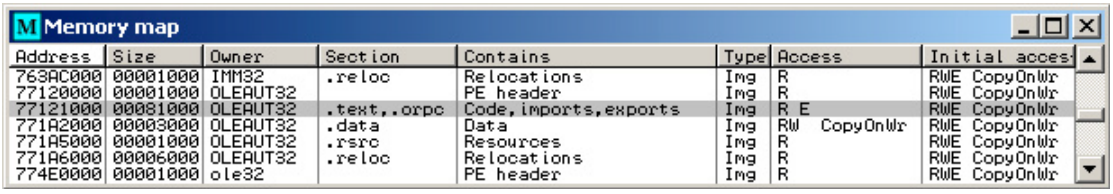
内存映射

信息概述

每一个 32 位的应用程序都会运行在它自己的虚拟 2^{32}-位内存空间。应用程序也只能获取低地址部分空间（2 或者 3 十亿字节）。Windows 系统用这个程序的可执行模块，数据块，栈和系统表。最小分配单元是一个页（4096 字节）。每个页都会有标识标明该页是否具有可读，可修改，可执行的属性。除了这一种保护措施，物理边界与内存块之间也没有其他的保护措施。如果程序分配了 2 个数据段，那么它们刚好会分配在相邻的位置上，然后应用程序也会把它两当成一个数据段。

Ollydbg 把应用程序的内存分布当成一系列独立的块。内存映射窗口显示调试器可以得到所有的内存块。同时这里也没有什么标准来决定块在哪里开始和在哪里结束，它可能发生在 OllyDbg 展示出几个已分配的内存块作为一个单独的内存块。但是在大多数情况下，明确的解决方法不是必要的。

在内存目录中的任何操作都可以限制块。在大多数情况下，这会使得工作容易而且促进调试。但是如果，举个例子，模块包含几个可执行的节（Sections），你将不会立刻看到整个代码分布。因此 OllyDbg 可能将相邻的块合成一个块，就如下面的一个例子：



系统 DLL 文件 oleaut32.dll 在 windowsXP 中申明了两个相邻的节：.text 和.orpc。他们有同样的属性，然后，OllyDbg 把这两个节解释为一个单独的内存块。

内存映射每一次暂停后都会更新。如果你需要在程序运行时实例化窗口，从弹出的菜单中选择**更新**按钮或者直接在内存映射窗口中按快捷键”**Ctrl+R**”。

双击内存映射中的一行来打开一个单独用来显示指向内存节目录块的窗口。

你也许为了一些指定的字节联系而在整个进程的内存分布中进行寻找（**Search…或者 Ctrl+B**，查看“搜索”章节下的细节描述）。当联系找到后，OllyDbg 打开 Dump 窗口然后

滚动下拉条到达最开始的发现的位置。按下 **Ctrl+L** 来发现下一个 Dump 位置或者按下 **Esc** 来关闭 Dump 窗口。在内存映射窗口中按下 **Ctrl+L** 来继续寻找。

内核存储

内核（系统的）内存被分配在高地址区域。通常它开始在地址为)0x80000000(家庭版本的系统)或者 0xC0000000（某些服务器版本的 Windows 系统）

如果 OllyDbg 可以访问到内核内存，它会使用一些特殊的调试函数来企图展示内核的详细信息。这些函数在 Windows XP 下很有效果，但在 Windows 7 下经常失败。像其他部分一样，OllyDbg 将内存状况显示为一个独立的快，其中不可用的内存页填充为零而不是问号。OllyDbg 既不能修改内核内存，也不能改变内核内存属性，同时也不能设置任何类型的断点。唯一可用的操作是搜索。注意，完整的查找会花费几分钟。

内核内存并不直接访问应用程序，而是由操作系统或者驱动来授权给这个内存部分。例如：一些杀毒程序会 LoadLibrary 和 GetProcAddress 等类似于内核内存的接口函数重定向到内核存储区域，在那里杀毒工具可以在不增加有效性的检查的情况下避免被病毒感染的危险。如果 OllyDbg 遇到一些函数，它会使用单步来进行跟踪。

备份

对于每一个内存块，除了内核，你都可以创建一个备份。这是一份当前内存目录的复制。如果备份是可用的，OllyDbg 将会高亮一些不同点。你可以浏览备份，重新存储修改处，存储备份到磁盘或者返回来读存储内容。如果你想要查看两个运行程序的不同，那么存储备份是非常有用的。

当你在 CPU 窗口编辑代码或者数据时，备份会自动生成。如果你要求当程序运行时，在修改处的命令暂停，那么备份是必须的。在节”Run trace and profiling”查看详细的描述。菜单 Option 下，**Debugging|Auto backup user code** 告诉 OllyDbg 在模块被加载到内存中时来创建可执行代码的备份。

注意独立的 Dump 窗口也许能创建他们自己的备份。他们从物理上来说和这里的备份是有本质的不同。

在内存入口中断

在内存映射窗口中，你可以在整个内存代码的入口点设置一次性断点。任何可能入口

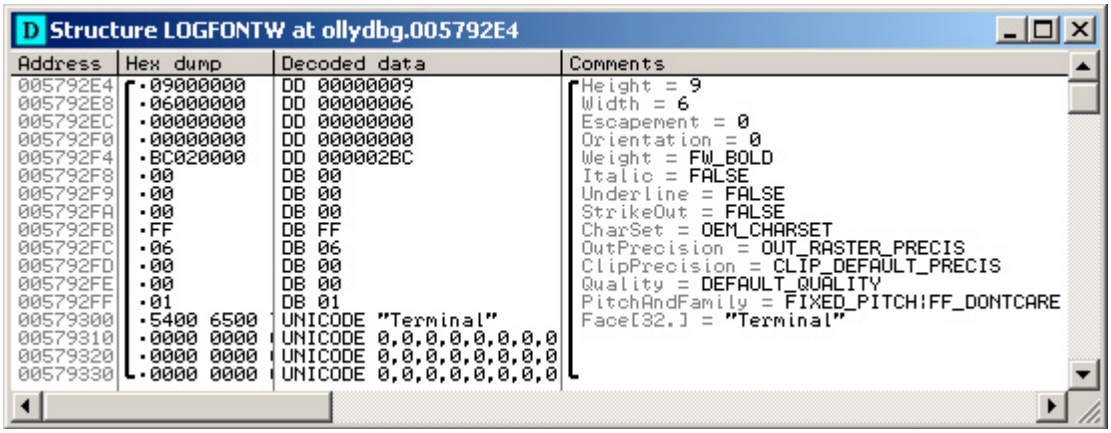
（读，写或者执行）将会触发中断并移除中断点。栈中的中断是不允许的，它也许会导致系统调用崩溃。

如果你需要找到调用 DLL 的函数或者这些函数的返回值，那么这种类型的内存断点是很有用的 。只需要在相应模块的代码区中设置断点即可。

Dumps[7]

Dump 窗口显示的是内存，文件，或者磁盘里的内容。CPU 窗口包含三种 Dumps: 反汇编器，Dump 和栈。一个窗口也可以打开无数个独立的 Dump 窗口。

创建一个Dump的方法有很多种。在内存映射中双击一行可以打开该选定的内存块的一个Dump。选定CPU反汇编器或CPU Dump的一块内存，然后选择菜单中的**Open in a separate Dump window（打开一个单独的Dump窗口）**命令可单独监视这块内存。如果你选择**Decode as structure...（按结构解码）**，这个选定的块会被解码成一个预定的Windows结构。如果你选择**Decode as pointer to structure...（按指针结构解码）**[8]，这个选定的块的在第一个字节开始的双字会被翻译成这个结构的指针，就像下面的例子中LOGFONTW转成CreateFontIndirectW()：

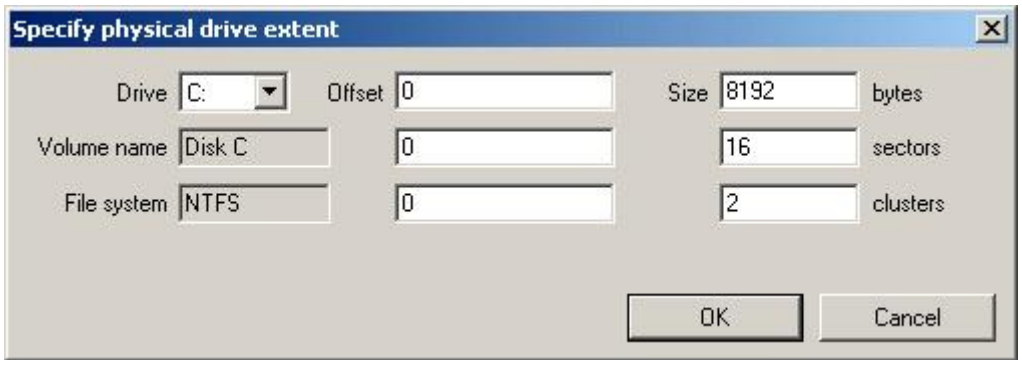


要打开一个文件，选择主菜单栏中的**View|File…（查看|文件）**。如果文件扩展名不是.exe，.dll和.link，你可以直接将它拖拽到OllyDbg里面。要保存修改后的文件，在弹出的菜单里选择**Save file…（保存文件）**。

[7] Dumps 的译文“转储”比较晦涩，所以文章沿用 Dumps——译者注

[8] 有用户称这两处选项在实际使用过程中没有找到——译者注

如果你有足够的管理者权限，你可以直接检查你的电脑的本地磁盘内容（**View|Drive…（查看|驱动）**）：



扩展的大小主要是被可利用的内存所限制的。驱动Dump是只读的，你不能保存任何修改到磁盘中去——这对于文件系统来说太危险了。

Dumps支持许多不同的格式：十六进制的ASCII，UNICODE或MBCS文本，纯ASCII或UNICODE文本，所有可能形式的16位和32位整数，堆栈视图，32位，64位和80位的浮点数，或者反汇编代码。对于ASCII和MBCS，你可以选择一个代码页安装在你电脑里。下面是一个多字节（变长）的UTF-8 Dump的例子：

“Variable-width(变长)”表示一个字符可以被一个，两个或多个字节替换，编码的长度取决于字符。OllyDbg支持每个字符最多五个字节，这满足所有当前存在的设置。注意上图中字符“O”在地址00418A3B处被编码成一个单独的字节4F，而从地址00418A42开始被选中的中国汉字需要3个字节：E4，B8，AD。

图中所示的Dump第一行包含10个UTF-8字符：7个拉丁（“OllyDbg”）和三个汉字。多字节的Dump有16个位置。6个未使用的位置在行的末尾用灰色的省略号填充，以此与明显的

空格区分开来。

OllyDbg在Dump中从不把十六进制表示的公认的多字节字符分行，所以一些Dump行中包含少于16个字节。

汉字，日本汉字或韩文符号明显比ASCII长，如果有人试图把它们强制转化成ASCII格式那么将会截去一部分。为了避免截去，激活选项**Dump|Use wide characters in UNICODE & multibyte Dumps(Dump|使用UNICODE长字符和多字节Dumps)**。ASCII和多字节代码页可以随时在选项中的**Dump**窗格或直接从弹出的菜单中被改变。

键盘上的方向键移动一次选择一个项目（十六进制Dumps的字节，UNICODE的长字符，反汇编器的命令。）按住Shift并移动选项可以选择多个项目。

组合键**Ctrl+UpArrow（Ctrl+向上键）和Ctrl+DownArrow（Ctrl+向下键）**滚动Dump窗口，但不根据项目的大小。这让你可以改变对齐方式。快捷键**Ctrl+LeftArrow（Ctrl+向左键）Ctrl+RightArrow（Ctrl+向右键）**可以使窗口按列左右滚动。

当选项**Dump|Underline fixups(Dump|强调固位)**激活，OllyDbg强调十六进制Dumps中的固位。固位是一个对可调整的地址的定位。如果可执行文件或DLL加载了一个预期外的地址，加载器就会改变地址的值。当你把代码放在一个被固位所占据的内存里时，要千万小心，尤其是DLL！OllyDbg将固位的内容调整回了默认的库，但是如果出于任何原因，模块加载在一个不同的库里加载了，代码就会发生改变（或者未固位的默认地址会指向旧的、现在无效的定位——我也不知道哪一种情况会更糟）。

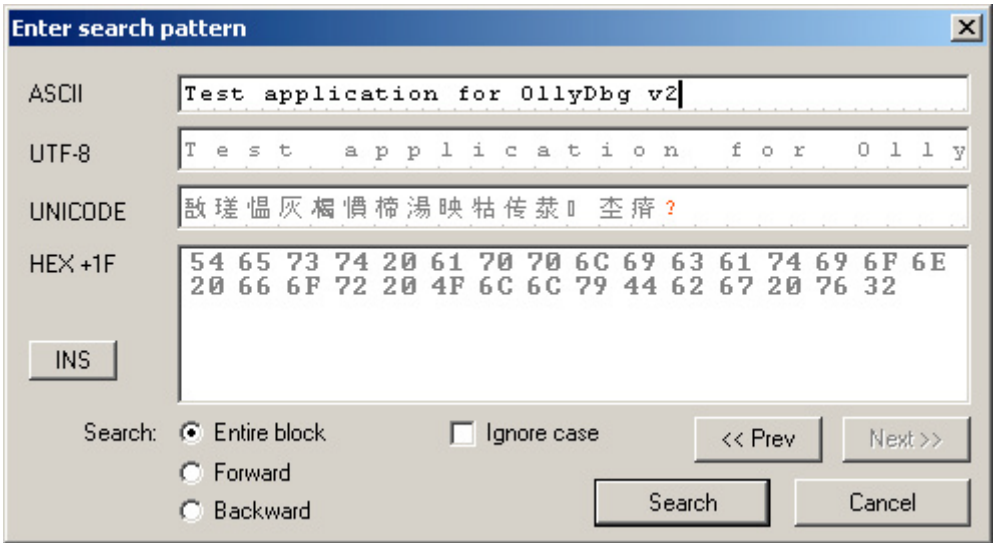
搜索

搜索二进制模式串

Ollydbg 提供了许多不同的搜索方案，该搜索样例就是搜索二进制模式串的。例如，要查找创建主要窗口的模式串在 Test.exe 中的位置，这个窗口的标题是**“Test application for OllyDbg v2”**。Test.exe 是一个 ascii 码的程序，我们希望名字也是 ascii 码编码的。一些编译器把静态数据加载到代码中，另一种则是使用数据段。我们之前不知道这个，所以必须就两种可能性一一排查。

切换到 CPU 反汇编，按下 **Ctrl+B**，或者在菜单中选择 **Search for | Binary string...（搜索->二进制串）**。在弹出的对话框中，你可以将字符串搜索指定为如下几种格式：

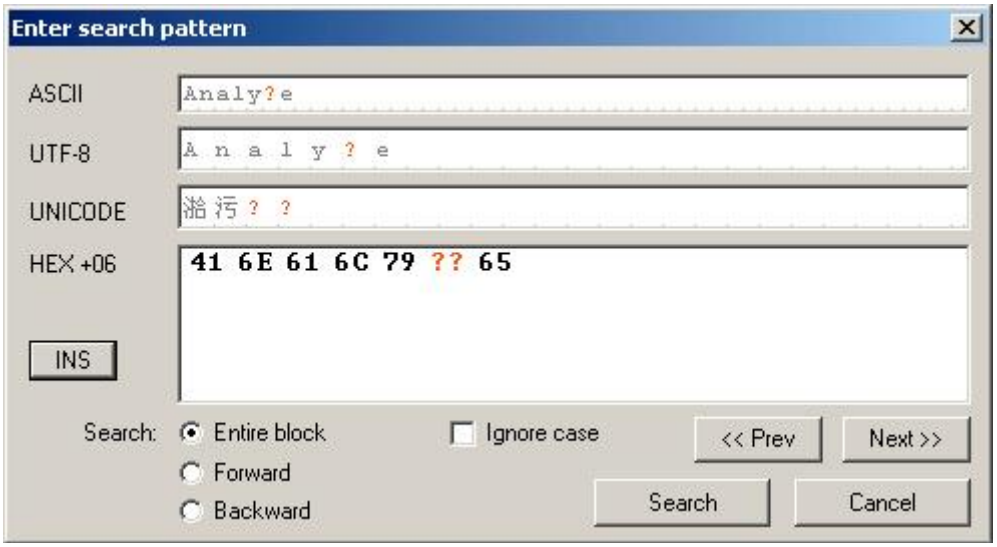
Ascii、多字节（MBCS）或者 UNICODE，或者直接设定为十六进制的字节序列。



ASCII 和 MBCS 使用当前的代码页，例如，在上文的例子中，ASCII 使用默认代码（1252，ANSI-Latin I），MBCS 则将编码设定为 UTF-8，点击上图中第一行（ASCII）并开始输入要查找的内容。随着输入数据的过程，Ollydbg 也会动态地更新其他行。凡是类似于 UNICODE 中的奇数字节这样无效的符号，都会显示红色的问号。

如果你在一行中选择文本的一部分，Ollydbg 会选择所有能够全部或者部分匹配上的字符。例如，如果你选择的第一个可见数字为 16 进制的 5，Ollydbg 将选择 ASCII5 的字母 T、MBCS（1 到 5 字节）还有第一个 UNICODE 的汉子字符。注意，Ollydbg 不支持字符替换。双字节的必须为 UNICODE 编码。范围在 D800-DFFF 的字符串要被视为独立的字符，即使他们并无实际意义。

16 进制的那一行支持模糊搜索，16 进制那一行的问号指的是忽略内容的半字节。例如，你想在程序中找到 Analyse（或者 Analyze）[9]这个字符串，在 ASCII 栏输入 Analyse，并选择字母 s，它的二进制码是 73，选择 16 进制编辑器。（**Ctrl+鼠标滚轮向上**）并用两个问号代替 73，这种方式就能搜索到英式和美式的 Analyse 了。



如果你粘贴到 16 进制控制栏，Ollydbg 会只显示 0-9 的数字以及 A-F 的字母，而忽略其他的字母。将“We apologize for the inconvenience”粘贴到 16 进制栏里，得出的结果是 EA EF EC EE CE 几个字符的组合。输入文本后，可以选择搜索方向：可以选择整个模块，也可以选择向前或者向后。**忽略大小写**的操作功能是十分有限的，它只针对标准的 ASCII 字符而忽略本地的字符。在遇到 MBCS 和 UNICODE 格式的字符时，忽略大小写功能可能会失效。

点击搜索按钮，不幸的是，在代码段中没有这样的字符串。在 CPU Dump 的窗口中选择第一个字节，然后按下 **Ctrl+L**，这个快捷键表示“继续在同一方向展开同样的搜索”（顺便说一下 Ctrl+Shift+L 表示在反方向上继续该搜索。）此时，文本在这里，为了确认，再按一次 Ctrl+L 键。状态栏提示**“Item not found”**则表示该字符串在文本中是唯一的。

不幸的是，这里有一个问题。我们已经找到了文本串，但是我们需要访问该命令所在的位置。别担心，有一个特殊的选项可以完成这样的功能。

搜索引用

对一些常量的引用就是一个包含这个常量的命令。例如，所有引用地址 00402007 的命

[9] “分析”这个词，英式英语和美式英语中有不同的拼法——译者注

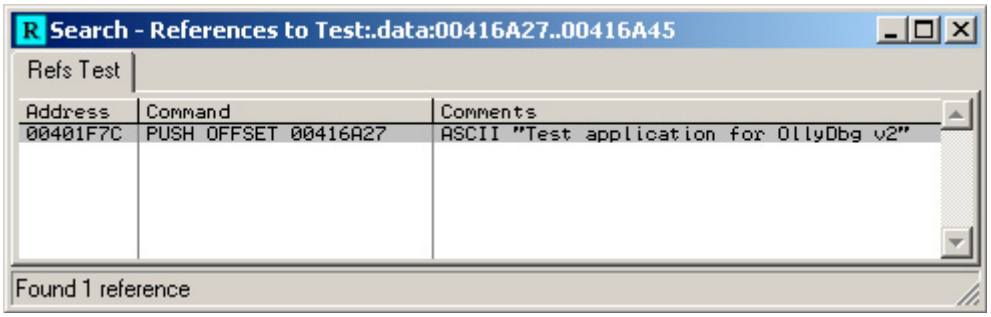
令：

MOV EAX，[00402007]

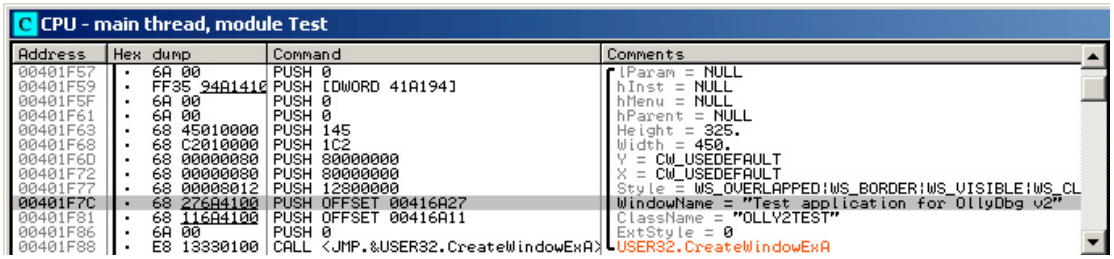
PUSH 00402007

CMP CL，[BYTE 00402007+ESI*4+EDI]

我们会发现在 CPU Dump 出来的文本中，这是“**Test application for OllyDbg v2**”的位置。选中这个位置，并在菜单中选择 **Find references to | Selected block**（查找|引用选块），搜索引用的命令就是这样。



双击这条索引，就能跳转到 CreateWindowEx（）调用。我们进入了创建窗口的主函数中。



有时索引并不一定会显示直接包含常量的命令，例如 PUSH　EDX。这不是一个错误，分析器试图传递函数的参数的值。命令的序列有时候很复杂，例如搜索 417634 会显示如下命令序列。

MOV EBX，　OFFSET 41751C

Long sequence oif commands that don't change EBX

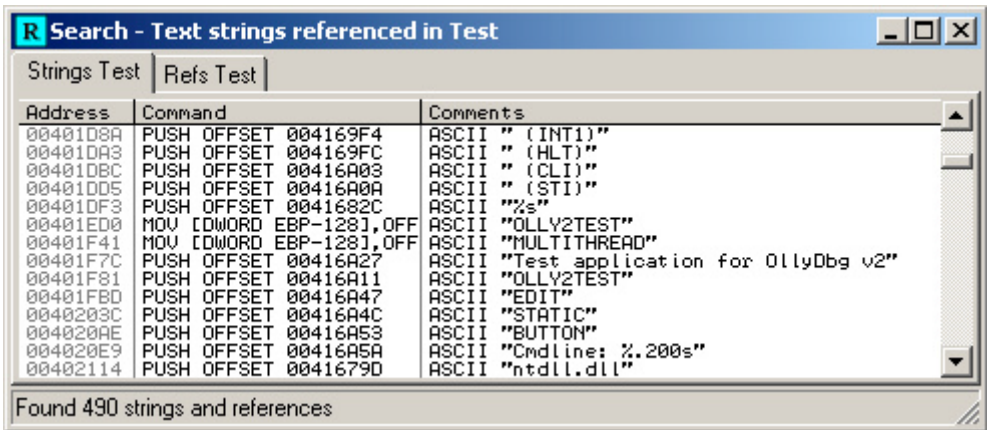
LEA EDX，[EBX+18]

PUSH EDX

EDX 包含的是 0041751c+0x18=00417534 这个值。

搜索引用的字符串

搜索引用的字符串，有一种更快地方法。同时它也适用于 ASCII 和 UNICODE 字符串。切换到反汇编的窗口，并在菜单中选择**“Search for | All referenced strings”**即可完成搜索。



点出的列表或许会很长，所以，搜索窗口会直接显示搜索内容（弹出菜单包括**文本形式的搜索列表、再一次搜索和逆向搜索三个选项**）。找到相应命令后，双击它就能跳转到反汇编窗口中该命令的位置处。

字符串的搜索利用的是分析器的结果，字符串采用启发式识别。默认情况下，ASCII 码被限制在拉丁文范围内，要允许国际字符，可以通过激活 **Strings | Allow** 选项来完成。请注意，这个选项不会考虑所选择的代码页。

如果 Strings | Allow 选项是激活的，UNICODE 格式的字符串和 ASCII 码格式的字符串限于同一个子集。Windows API 自带的启发式分析器 TextUnicode()可以分析许多其他的语种。**(Strings | Use 是 TextUnicode()选项)**。但是请注意，这个解码函数可能返回错误的结果。甚至把独立的字符和 ASCII 码解码成 UNICODE。

字符串一般都以空字符结尾，有些编程语言，如 Pascal 和 Delphi 可能会使用另一种形式。其中，ASCII 串受字符个数影响，Pascal 串的识别有字符串本身影响。

请注意，如果你改变了上面提到的设置和选项，你必须在反汇编窗口中键入 **Ctrl+A** 来重复分析过程。

常量搜索

你可以直接或者间接地找到包含特定常量的命令。它和索引很像，但是常量搜索不局限于地址。例如，搜索常数 1，会得到

PUSH 1

SHL EAX，1

MOV [BYTE EAX+1]，4567

DD 00000001

这样的指令以及其使用常数 1 的指令。

单个命令搜索及一系列命令的搜索

你可以在 Ollydbg 上搜索一条命令，也可以搜索一个命令的序列。这种行为很频繁，因为 80x86 架构提供很多重叠的寻址模式，且很多操作码表示的是相同的命令。举个例子说，MOV EAX，[EBX].这条命令有 16 个可能的二进制编码：

8B03 – 最简单的形式

8B43 00 – 带有一个字节位移的非SIB形式

8B83 00000000 – 有四个字节偏移的非SIB形式

8B0423 – 没有标度指数的SIB形式

8B0463 –同上

8B04A3 – 同上

8B04E3 – 同上

8B4423 00 – 一个字节偏移、无索引的SIB形式

8B4463 00 – 同上

8B44A3 00 – 同上

8B44E3 00 – 同上

8B8423 00000000 –四个字节偏移、无索引的SIB形式

8B8463 00000000 – 同上

8B84A3 00000000 – 同上

8B84E3 00000000 – 同上

8B041D 00000000 – 四字节偏移，标准比例，没有基地址

由于每条命令前都可以加前缀 DS（操作码 3E），所以命令条数加倍。

但是这不是 Ollydbg 本身的问题，无论有没有前缀，Ollydbg 都会找到所有的命令（如果你只想查找带前缀的命令，那么在搜索模型中指定该前缀）。

命令中可能含有不准确的操作数，比如任何一个 32 位的寄存器，任何内存地址或者任何操作数。例如，MOV EAX，ANY 会和 MOV EAX，ECX，MOV EAX，12345;，MOV EAX，[FS:0]等很多命令进行匹配。

不精确的使用下列伪操作数：

关键字	对照
R8	任意八位寄存器 (AL，BL， CL， DL， AH， BH， CH，DH)
R16	任意16位寄存器 (AX， BX， CX， DX， SP， BP， SI， DI)
R32	任意32位寄存器 (EAX， EBX， ECX， EDX， ESP， EBP，ESI， EDI)
SEG	任意段寄存器 (ES， CS， SS， DS， FS， GS)
FPUREG	任意浮点运算寄存器 (ST0..ST7)
MMXREG	任意增强指令寄存器 (MM0..MM7)
SSEREG	任意指令集寄存器 (XMM0..XMM7)
CRREG	任意控制寄存器 (CR0..CR7)
DRREG	任意调试寄存器 (DR0..DR7)
CONST	任意常量
ANY	任意寄存器、常量或者操作数

你可以对内存地址的这些关键字进行自由组合，具体示例如下：

内存地址	对照
[CONST]	任何固定的内存为止，比如 [400000]
[R32]	寄存器中存储的地址，比如 [ESI]
[R32+1000]	32位寄存器的值与0x1000的和，比如[EBP+1000]
[R32+CONST]	32位寄存器与偏移的和，比如 [EAX-4] 或者 [EBP+8]
[ANY]	内存操作数，比如[ESI] 或 [FS:EAX*8+ESI+1234]

如果你在寻找命令的序列， 命令在序列中的相互影响是很重要的。假设你在寻找两个操作数的比较。而 8086 架构没有这样的指令（除了 CMPS 指令，但是该指令执行缓慢，需要长期的准备）因此，编译器将产生两个命令：

MOV EAX，[DWORD 408804]

CMP EAX，[DWORD 408808]

然而，编译器用 ECX 或其他寄存器代替 EAX 也是有可能的。考虑到这些情况，Ollydbg

提供了具有特殊依赖性的寄存器。

寄存器	含义
RA， RB	**RA** 与 **RB**必须类型相同，同为32位寄存器，但是**RA** 与 **RB**不同
R8A， R8B	同上， **R8A**和 **R8B** 都是8位寄存器
R16A， R16B	同上，但是 **R16A** 和 **R16B** 都是16位寄存器
R32A， R32B	与 **RA， RB** 要求相同

因此，搜索 XOR RA，RA，我们会找到异或操作的所有 32 位寄存器，XOR RA，RB 则会排除这种情况，上述例子的正确写法如下：

```
MOV RA，[CONST]
CMP RA，[CONST]
```

优化编译器可以在其中插入其他的命令。

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

命令 MOV ESI，401007 和 SUB ECX，EAX 不改变 EAX 寄存器的内容 SUB ECX，8 修改了 EFL，但 CMP EAX，[DWORD 408808]不使用标签。因此，两个附加的命令对内存比较没有影响。如果允许查找中间命令的操作被激活，那么他们将被忽略。当然，系统调用和无条件跳转是不允许的。

命令的功能性是可以被外部跳转影响的。如果想要排除这样的序列，那么，请取消允许外部跳转的选项。在其他的搜索框里，其它选项的操作也是一样的。

也有一些不准确的命令：

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

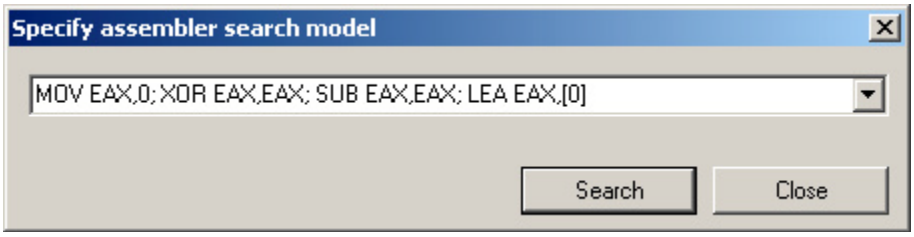
举例：

格式	命令
MOV R32，ANY	MOV EBX，EAX
	MOV EAX，ECX
	MOV EAX，[DWORD 4591DB]

	MOV EDX，[DWORD EBP+8]
	MOV EDX，[DWORD EAX*4+EDX]
	MOV EAX，004011BC
ADD R8，CONST	ADD AL，30
	ADD CL，0E0
	ADD DL，7
XOR ANY，ANY	XOR EAX，EAX
	XOR AX，SI
	XOR AL，01
	XOR ESI，00000088
	XOR [DWORD EBX+4]，00000002
	XOR ECX，[DWORD EBP-12C]
MOV EAX，[ESI+CONST]	MOV EAX，[DWORD ESI+0A0]
	MOV EAX，[DWORD ESI+18]
	MOV EAX，[DWORD ESI-30]

需要注意的是：在最后一行的[DWORD ESI-30]与[DWORD ESI+0FFFFFFD0]是相等的

在搜索模式的每一行中，都可以用分号分隔多个命令。如果你想要找到 EAX 置零的所有命令，可能的解决方案如图所示：



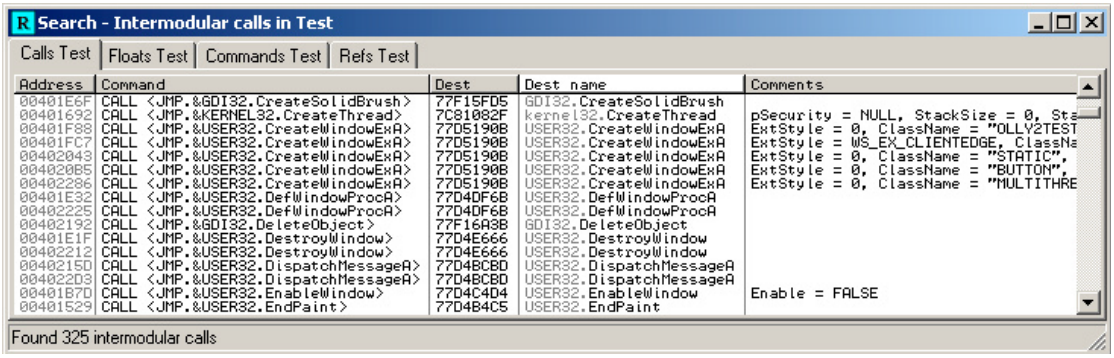
如果你对此感到好奇，那么，缺少的指令有 AND EAX，0；MUL EAX，0；MOV EAX，[zeromemory]。和 SHL、EAX，0x20。根据指令的要求，32 位的模式下，移位的位数被限定在 5 位，所以搜索器可能不会如我们所愿的正常工作。

搜索所有项目

上文中描述的许多搜索类型都存在两个版本:

1. 查找下一个项目
2. 查找所有项目

Ollydbg 能够记忆最多八个之前的搜索记录，这些搜索记录选项由搜索函数命名（Calls 指的是模块调用，Commands 指的是单个或者多个命令，REFS 则指的是一系列的参考等等）。函数名在模块名之后。



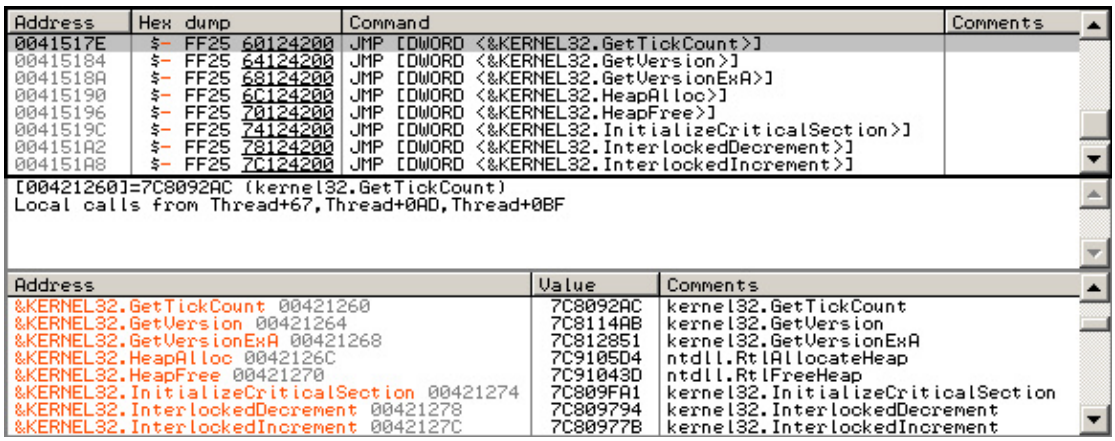
如果在窗口上已经有 8 条记录，新的搜索将关闭最右边的记录选项卡。如果想保留它，可以用鼠标右键点击选项的名称，选择将此**选项前置**。如果用鼠标中间的滚轮点击该选项，该选项就会关闭。

在某一行的位置上双击，就能弹出反汇编指令。快捷键 **ALT+F7** 和 **ALT+F8** 能显示出所有的位置顺序信息。顺便说一下，这些快捷键在反汇编窗口中同样适用。

搜索所有跨模块调用

这种搜索方式试图找到某一模块在不同的模块中被调用的所有位置。上面的截图里就有一个例子。显示内容的一栏列出了传递给系统调用的参数。

要查找函数，先输入函数名称（不带模块名）。搜索窗口滚动到第一个一输入的函数名为开始的函数的位置上。输入函数名不区分大小写。有几种调用导入函数的方法：Borland 编译器加载系统来存储导入函数列表的真实地址（通常是在.data 段中，请看下文中的 Dump）在代码段的结尾，每一个导入函数都会生成一个间接跳转。调用导入函数实际上是间接跳转到导入函数的入口处。



如果你想拦截特定模块 API 函数的所有静态调用，跳转到导入地址通常是很好的方法。

请注意 Ollydbg 用来描述这些项的名称。导入函数的地址是模块函数的地址。导入函数表的内存地址包含了函数引用的地址。为了从函数本身的地址中区分出函数引用的地址，Ollydbg 预先将函数名中添加一个符号&。调用一个函数实际上就是调用跳转到该函数地址的指令，但是这些跳转指令没有特别的标签。为了强调间接跳转的事实并设置目的地址，Ollydbg 预先考虑将 JMP 和目的地址放在尖括号中，形如<JMP.&module.function>。最终反汇编的跳转指令会变成形如 CALL <JMP.&KERNEL32.GetModuleHandleA>的样子。

顺便说一下，你注意到 HeapAlloc 的地址被标记成&KERNEL32.HeapAlloc。但是却显示为 ntdll.RtlAllocateHeap 了么？这就是一个出口转发的例子。由于历史原因，HeapAlloc()这个函数是由 kerne32.dll 声明的。但是它的功能等同于 RtlAllocateheap（）。它为编写一个将调用 RtlAllocateHeap()简单跳转到使用同一参数的另一个函数的存根提供了可能。但是，为了分享链接和执行时间，Kernel32 必须告诉加载器，他必须直接调用 ntdll。对于一个简单可理解的说明，我建议读者看一下由 Matt Pietrek.编著的 Win32 的 PE 文件格式的第二部分。

微软编译器在跨模块调用的处理上有一点不同，存放导入目的地址的表通常在代码段的起始部分。且每一个函数的调用都只是通过导入表间接调用。下面是一个例子：

CALL [DWORD <&KERNEL32.GetModuleHandleA>]

GNU 编译器两种调用方法都使用，甚至是在一个单独的可执行文件当中也是如此。

对系统调用的搜索使用分析器来进行预测，如果 Ollydbg 告诉你命令 CALL　EDI 是一个 GetModuleHandleA()的调用，那么前面一定有一个 MOV EDI，[DWORD <&KERNEL32.GetModuleHandleA>]作为调用之前的声明。

线程

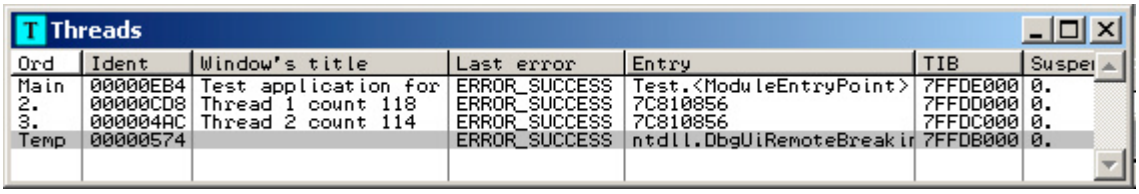
信息概述

OllyDbg 可以调试多线程应用程序。

线程通常没有名字，当新线程被创建后，窗口会分配一个系统标示符给它。不过遗憾的是，每个应用程序启动时这个标识符都是不同的。

为了解决这个问题，当线程被创建时 OllyDbg 会列举全部线程。主线程调试器默认序号 1，之后的线程被编号为 2，3 等等。若应用程序顺序创建固定的线程，序号会在调试时保持表述一致。标示符和序号表述都支持。

有时操作系统会创建自己的线程，例如，如果你 DebugBreakProcess()暂停调试应用程序，OllyDbg 的标签是暂时的并且不会分配序号：



在线程窗口中由线程创建的窗口内容标题会在纵列的标题窗口上置顶。如果线程创建多个窗口，OllyDbg 将随机选择其中的一个置顶。

纵列的最近一次错误窗口上显示了最新的错误代码集，是 GetLastError()的返回值。

你可以手动挂起和恢复线程，注意如果 OllyDbg 脱离调试状态手动挂起线程是无法恢复的。

我刚才提到的线程通常但并非总是无名的。有一种特殊的、由视觉组件使用的服务异常 0x406D1388 (MS_VC_EXCEPTION)，它由 Ollydbg 所支持，并向调试器传递线程名。你也可以命名线程（在线程窗口的弹出菜单中选择 **Set symbolic name**），线程名在表述时不被保留。

步进多线程应用程序

当调试多线程应用程序时有一个你必须注意的警告。各种步进，像单步步过、运行跟踪、hit 跟踪或执行直到当前线程执行返回。试想一下下面的场景：xxx 函数给不同的线程发送信号并等待响应。你单步步过调用 xxx，OllyDbg 挂起除当前线程外的全部线程，通过指令设置临时断点后继续执行和调用。xxx 发送信号，等待响应然后没有任何响应发生，因为线程所在的这个进程是暂停的。如果发生这种情况，暂停程序并以全部线程模式运行它，或

手动重新调换其它不同的线程。

异常处理

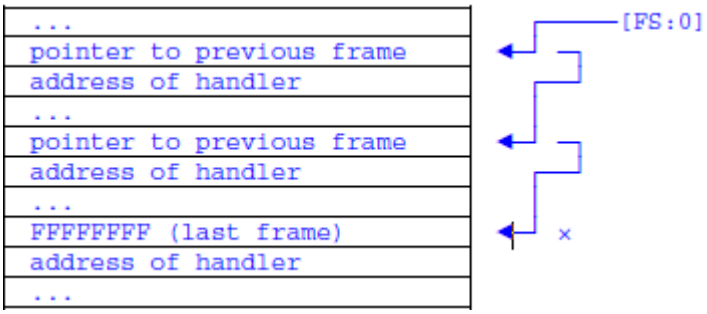
这儿有两种异常处理机制与窗口成为一体：结构化异常处理(SEH)和向量化异常处理(VEH)。在这两种情况下，异常处理程序被组织成链条。

SHE 处理是线程依赖，每个线程必须单独安装它自己的处理程序。SHE 链的链接被称为帧并保存在栈上。最新安装的帧指到[DWORD FS:0]位置。(纸面上 Win32 内存模型选择器 CS，DS，ES 并且 DS 覆盖整个逻辑内存且基址为 0。FS 是一个异常，它的基址相当于线程的数据块的地址，也被称为线程信息块的地址，或 TIB)。

SHE 框架大致相当于 C++中的_try…_except 模块。当编译区遇到_try 时，它会插入

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

或与上述序列相对等的指令序列。在安装了几个处理程序之后，堆栈结构如下：



要删除最近安装的处理程序，线程可以完成如下指令：

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

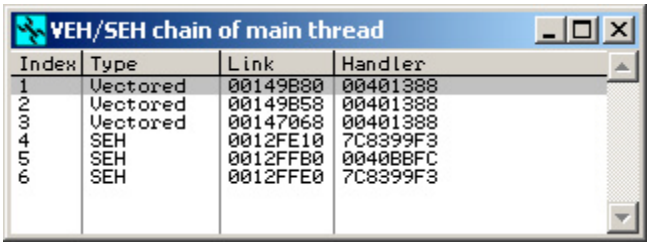
当然，寄存器可能是不同的。用 POP ECX 代替，编译器会发出 ADD ESP.4;用[DWORD FS:0] 代替[DWORD FS:EAX]且先于序列 XOR EAX，EAX 等等。

当异常出现时，Windows 遍历链条并且询问每个处理程序是否可以处理此事件。如果处理程序返回 EXCEPTION_CONTINUE_SEARCH， Windows 尝试继续处理等等，直到处理程序发现或者走到链条结束符，在这种情况下 Windows 调用未处理的异常处理过滤器。(默

认 UEF 通常终止应用程序)。

向量化异常处理在 Windows ME 或 Windows 2000 下不可用。向量化处理器优先于结构化处理器且与线程独立。进程可以将处理程序添加到队列的头部或尾部。如何管理这个队列是没有记录的。实际上，Windows 试图通过使用进程唯一键和不可用的队列的确切位置来干扰隐藏处理程序的地址。**因此只有当进程被 OllyDbg 自身启动时并且只有当选择调试或在系统中设置一直活跃的永久断点，OllyDbg 才能够列出 VEH 处理程序**。同时，它可能有新的服务包出现 OllyDbg 将失去追踪 VEH 链条的能力。

现在让我们做一个快速的实验。在 OllyDbg 中加载 Test.exe，运行它并按下“Set VEH”几次，暂停应用程序并从主菜单中选择查看或 VEH/SHE 链条(或按 Alt+S)：



当异常发生时，该处理程序将被列进它们将要调用的列表(处理程序索引是第一个为 1)。

如果你在独立模式中且附着于 OllyDbg 上之后启动 Test.exe，向量处理程序的地址将被标记为无效。

表达式和监视窗口

信息概述

OllyDbg 在监视时使用表达式、条件断点和运行跟踪条件。赋值表达式分两步完成。首先，OllyDbg 编译成中间二进制的形式表达。第二步，计算它的值。赋值通常比编译要快几个数量级，这对断点和跟踪至关重要。

表达式大致可以分为两类：一类用于作出决定(例如，条件断点)，另一类用于显示数据。第一种情况你只需要一个值。第二种情况，你可以一起看到多个值和表达式。

在内部，浮点数有 80 位精度。所有其他元素被保持为一个 32 位的整数数字。

基本元素

表达式会包含：

- 字节寄存器 AL， BL， CL， DL， AH， BH， CH， DH
- 字寄存器 AX， BX， CX， DX， SP， BP， SI， DI
- 双字寄存器 EAX， EBX， ECX， EDX， ESP， EBP， ESI， EDI
- 段寄存器 CS， DS， ES， SS， FS， GS
- FPU(浮点运算单元)寄存器 ST0， ST1， ST2， ST3， ST4， ST5， ST6， ST7 或替代形式的 ST， ST(0)， ST(1)， ST(2)，ST(3)， ST(4)， ST(5)， ST(6)， ST(7)
- 索引指针 EIP
- CPU 标志 EFL 或另一种形式的 FLAGS
- SSE 寄存器 MXCSR
- 简单的标签，像 GetWindowTextA 或 userdefinedlabel
- 已知常量，如 WM_PAINT 或 ERROR_FILE_NOT_FOUND
- 把预先计划好的模块名称贴上，像 user32.GetWindowTextA
- 立即整数，像 ABCDEF01，123，0x123(所有十六进制)或 123.(十进制)
- 立即浮点数，像 123.456e-33
- 字符串常量，如“string”(用于比较)
- 参数%A 和%B
- 线程标示符%THR(或另一种形式%THREAD)
- 当前线程的序号%ORD(或 alternative from %ORDINAL)
- 内存的内容(需要方括号[]，参见下面详细说明)

元素按列出的次序处理。因此 AH 总是被理解为一个 8 位寄存器，而不是一个 16 进制常量 0AH。出于同样的原因，标签 EBP 不能再表达式中使用(但 mymodule.EBP 就可以)。如果有一个名为 ABCD 的标签，ABCD 这个标签将被解释为地址，而不是十六进制数 0x0000ABCD。

存储器内容

要访问存储器，将地址放到方括号中并选择指定该项目的类型和大小。OllyDbg 支持下列修饰符：

Modifier	如何理解存储器内容
BYTE	8 位无符号整数
CHAR	8 位有符号整数
WORD	16 位无符号整数
SHORT	16 位有符号整数
DWORD	32 为无符号整数(默认)
INT， LONG	32 为有符号整数
FLOAT	32 位浮点数
DOUBLE	64 位浮点数
LONG DOUBLE	80 为浮点数
ASCII	在括号内的地址将被解释为一个指向 ASCII 字符串的指针，但表达式的数值不变
UNICODE	UNICODE 字符串，但数值表达式保持不变

语法规则相对宽松。以下五个表达式具有相同的含义：

```
[400000+EAX*32]                ;需要注意的是DWORD是缺省的
DWORD [400000+EAX*32]
[DWORD 32*EAX+400000]
DS:[DWORD 32*EAX+400000]
[DWORD DS:32*EAX+400000]
```

存储器内容可以用于存储另一个存储器地址。假设 ppi 被声明为 int ** ppi(导出名是_ppi)且我们想监控**ppi 的整数值。正确的表达式是：[INT [[_ppi]]]

需要注意的是，我们需要三对括号。具体为：

_ppi　　是变量的地址 (*&ppi*)

[_ppi]　是变量ppi所指内容的值

[[_ppi]]　　是*ppi的值，ppi所指内容指向的地址

[[[_ppi]]]　ppi所指内容指向的地址空间的值

[INT [[_ppi]]] 要求 OllyDbg 将 ppi 所指内容指向的地址强转为整型

举一个例子，我们有矩阵 rect[4][4]并且向了解 rect[2][3]的值。结构体 RECT 如下：

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

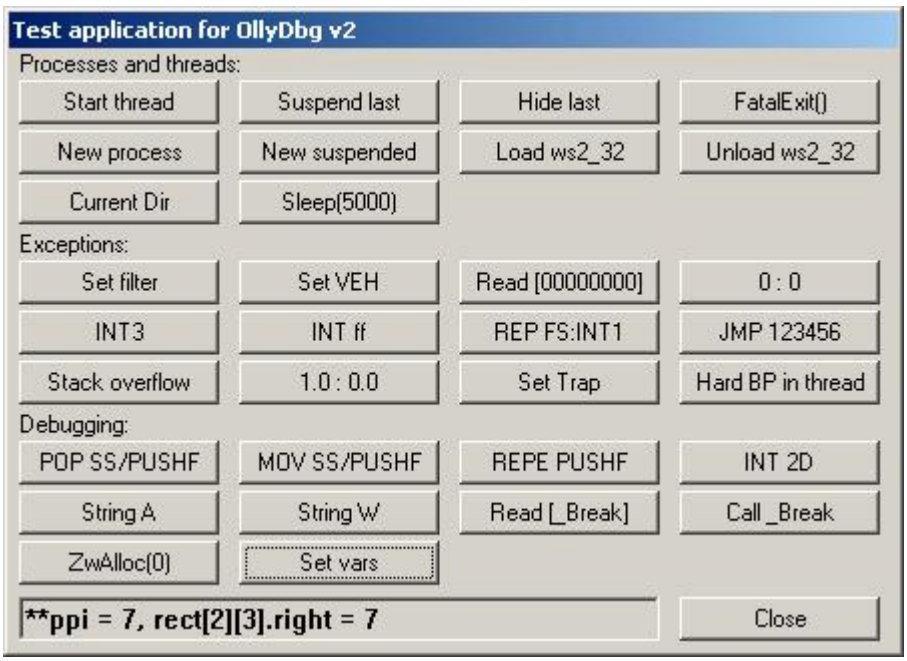
首先，RECT 结构体的大小是 4*sizeof(Long)=16 字节。_rect 是 RECT 四部分的首地址。为了得到 rect[2]，我们必须通过 2*4*sizeof(RECT)字节推进地址：_rect+2*4*16。为了得到 rect[2][3]，我们必须通过结构 3 移动指针：_rect+2*4*16+3*16。现在我们需要右移，即 RECT 结构体里的第三个双字：_rect+2*4*16+3*16+2*4。我们已经计算出 rect[2][3].right 的地址。为了得到所指存储器的内容，我们必须将这个地址放到方括号中。最终表达式将是

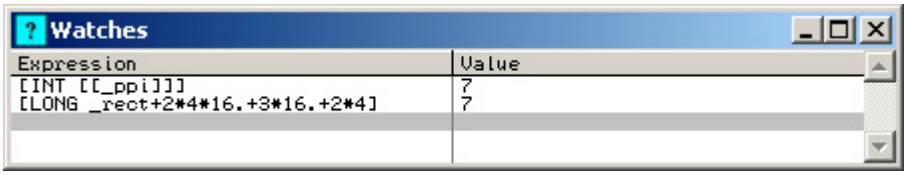
[LONG _rect+2*4*16+3*16+2*4] – 错误表达式!

等一等，你发现错误了吗？默认情况下，OllyDbg 全部被解释为十六进制的数字，因此 RECT 结构体有效的大小为 16(16 进制)=22(10 进制)字节，这是不对的。那应该怎么做呢？如果是十进制数，追加小数点。正确的表达式是

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

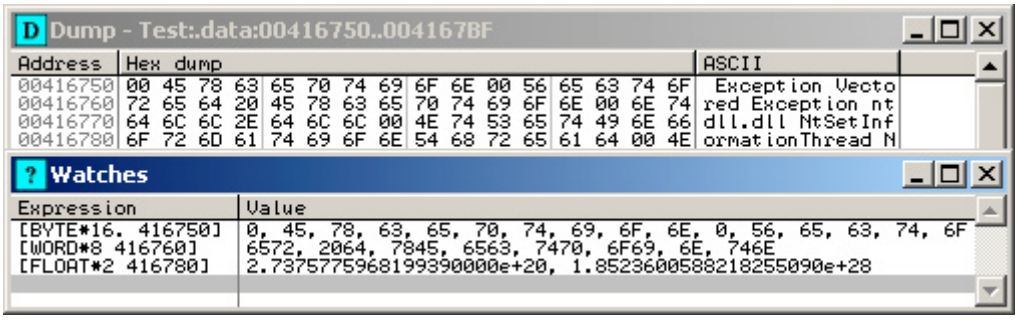
常数 2，3 和 4 是相同的基准，因此没必要用小数点。如上所述 Test.exe 声明_ppi 和 _rect。要改变这些变量，点击按钮“set vars”：





注意监视窗口只对某些事件更新，例如执行暂停或重绘。如果你需要不断监测变量，请在弹出式菜单中激活**外观|自动更新**。如果**自动更新**是激活的，OllyDbg 将定期更新受影响的窗口内容。默认间隔为 1 秒。可以在选项中进行更改(**外观|自动更新间隔**)。

你可以通过指定重复计数显示 16 个连续的寄存器位置。它只允许最外面的寄存器和[modifier*repcount address]形式。修饰符是强制性的，单独的重复计数如[*10 ESI]被视为语法错误：



有符号数据和无符号数据

默认情况下，所有的整型变量和常量都是无符号的。浮点项目通常是有符号的。以翻译项目作为有符号的，预先附加一元符号(‘+’或‘-’)或者附加 SIGNED 修饰符。从有符号转为无符号，使用 UNSIGNED。例如：

无符号	有符号
0xFFFFFFF	+0xFFFFFFF(-1.)
EAX	+EAX
ECX	SIGNED ECX
[DWORD 4F5024]	[SIGNED DWORD 4F5024]

运算符

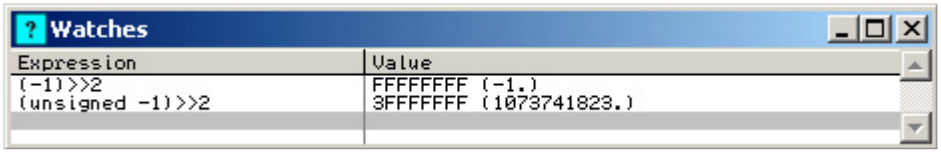
算术，逻辑和比较运算符支持通过 OllyDbg 形成一个 C 语言操作符和类似优先级的子集：

操作符	含义
+ - ! ~	一元加，算术否定，逻辑否定，求反（优先级最高）
*** / %**	乘法，除法，余数
+ -	加法，减法
<< >>	左移，右移
< <= > >=	小于，小于或等于，大于，大于或等于
== !=	等价，非等价
&	按位与
^	按位异或
|	按位或
&&	逻辑与
||	逻辑异或

复杂的表达式和嵌套的括号和寄存器括号是有最大长度限制的（255 个字符）。

如果二进制操作符的两个操作符不同类型，OllyDbg 将它们转为更高的一组类型：（浮点型，无符号，有符号）。唯一的例外是位移（<<和>>）：如果左移的有符号操作符和右

移的无符号操作符，OllyDbg 会使用有符号位移且结果也是有符号类型的。看一下：

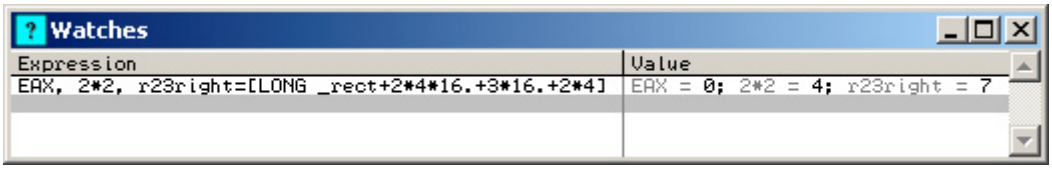


在表达式(-1)>>2，-1 是一个有符号类型的数字(按照上面的规则)它的二进制的值 0xFFFFFFFF。

因此 OllyDbg 使有符号位移，且每个位移循环最高有效位设置为 0。

复杂表达式

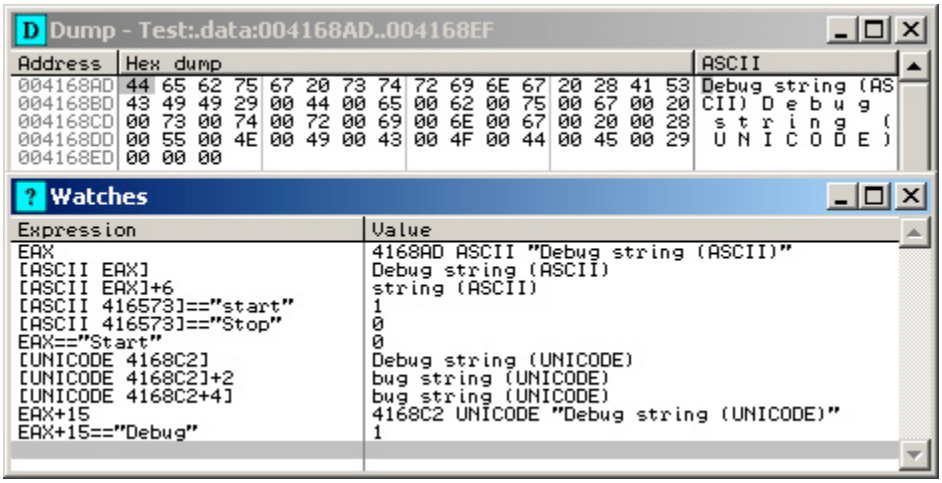
如果表达式仅仅用于显示数据，它可以包含几个表达式的形式 text1=expr1，text2=expr2…字表达式的数量限制是 16 个。表达式（text1，text2）是在赋值过程中忽略的，只标注结果。如果表达式不存在，OllyDbg 会自动的创建一个。例如:



字符串操作符

表达式可以包含字符串和字符串常量。字符串有[ASCII address]或[UNICODE address]两种形式，地址也是一个表达式，它指定第一个字符的地址。

字符串的操作符是被限制的。你可以在字符串中加或减去整数（C 语言规则应用）:[UNICODE EAX]+2 相当于[UNICODE EAX+4]，是因为一个 UNICODE 字符的长度是 2 个字节。你可以比较字符串与字符串常量，忽略情况：[ASCII ESI]==“ABC”。比较对字符串常量的长度限制，如果 ESI 指向“ABCDE”、“aBc”或“abc---”那么这个表达式为真，如果 ESI 指向“AB”或“AABC”则为假。你也可以比较字符串常量和指针 ESI==“ABC”。这样比较是非常类似于前面的，所不同的是你不需要指定字符串类型-OllyDbg 首先假定 ESI 指向 ASCII 字符串，且如果比较失败，尝试 UNICODE。如果 ESI 指向 ASCII “Abcde”或 UNICODE“abc”，表达式 ESI==“ABC”为真。



分析

OllyDbg 是一个分析调试器。对于每一个模块（可执行文件或者 DLL）它都会试图将代码从数据中分离出来，识别过程，定位嵌入的字符串和转换表，确定判断和循环，找到函数调用并解码，甚至在执行过程中预测寄存器的值。

这个任务并不简单。现有的一些编译器是高度优化的并且使用不同的方法加速。所以不可能把它们所有的都考虑在内。因此，分析器不能局限于一些比较流行的编译器而是尝试使用一种可以对任意代码都很好的工作的通用的规则。

这怎么可能呢？OllyDbg 制作了 12 个 passes 来通过程序，用来收集下一步执行的信息。例如，第一个 pass 反汇编了代码段的每一个可能的地址并且检查是否可以是指令的第一位。

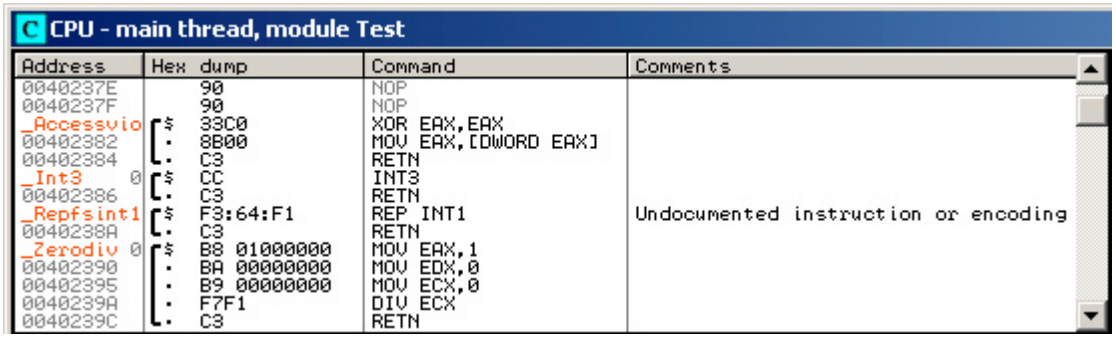
有效的指令无法启动修复，它不能跳到不存在的内存。此外，它会对每一个函数调用进行计数。当然，有一些函数调用时人为写的，但是不会发生两个错误的函数调用指向同一条指令的情况，也就更不可能有 3 个的情况。所以，当有 3 个或者更多的函数指向同一个地址，分析器就会确定这个地址是一些被频繁调用的子程序的入口点。第二种方式，OllyDbg 会用找到的输入作为入口点来确定其他的输入。以这种方式，99.9%的指令可以被确认。当然，一些字节不在这个范围内。我运用了 20 种高效的启发式方法来定位它们，但是这些方法都不可靠，而且分析器会出现错误，不过每次释放都会使错误的数量减少。

流程

流程的定义是一个连续的，从入口点开始，在理论上来讲，可以到达所有指令（除了 NOP 或者类似 spacehplders 那样填补的指令）的代码段。严格的流程有一个入口点和至少一个返回值。如果你选择 fuzzy 模式，或多或少不太一致的地方都会被视为单独的流程。

现代编译器会进行全局代码优化，这样可以在很多组成的时候拆分过程。在这种情况下，fuzzy 模式就会非常有用。但是，分析错误的可能性依然很高。

在 dump 栏流程会被大括号标记。美元符号（$）标记函数调用的目的地址，符号“大于”（>）标志跳转的地址，点（.）标志其他的分析指令。下面图片中的四个流程是**"Read [000000]"， "INT3"， "REFS:INT1"** 和 **"0 : 0"**四个按钮在测试程序 Test.exe 中使用后的功能：



栈变量

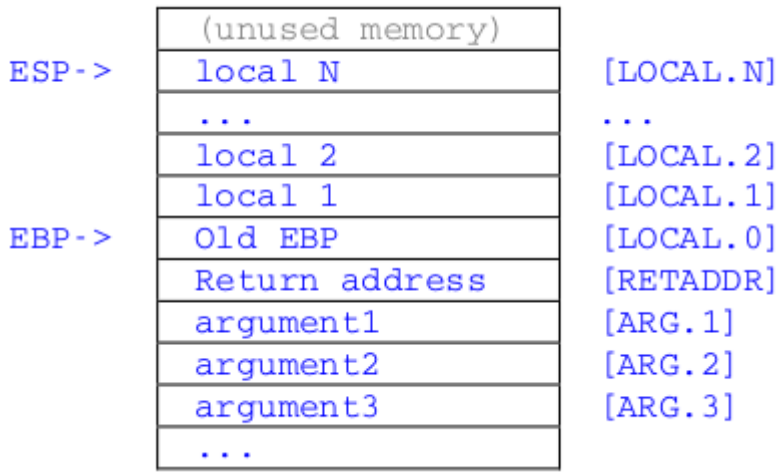
函数调用时传入栈变量参数的情况通常看起来是这样(假设都是双字参数):

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

被调用函数创建了新的栈帧(但并不总是这样!)并在栈上的本地内存中分配了 N 个双字空间:

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

在上面这些序列执行完毕之后，栈的布局如下:



ARG.1 标记了栈中第一个函数的地址，[ARG.1]是它的内容.Arg.2 是第二个参数的地址，

依次类推.LOCAL.0 是返回地址之前的双字立即数.

如果被调用函数创建了标准栈帧，那么[LOCAL.0]保留的是上一次的 ESP 值，局部数据从 LOCAL.1 开始，反之，局部数据从 LOCAL.0 开始.

注意 ARGs 和 LOCALs 都是十进制索引-由小数点符号指出.

一些子例程会篡改栈上的返回地址.如果 OllyDbg 检测到了对这个位置的访问，它将会对其标记以[RETADDR].

当子例程使用标准栈帧，寄存器 EBP 作为一个帧指针来使用.LOCAL.1 是 EPB-4，LOCAL.2 是 EPB-8，ARG.1 是 EPB+8 等等.现代优化过的编译器更喜欢使用 EBP 作为通用寄存器而使地址参数和局部变量处于 ESP 之上.当然，它们必须记录下所有对 ESP 的修改.看一下下面的代码:

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

当过程被调用时(地址为 F:)，ESP 指向栈上的返回地址.下面的两个双字是第二个参数 ARG.2.命令 PUSH 将会将 ESP 减去 4.因此最后一行访问的是 ARG.1.代码等价于:

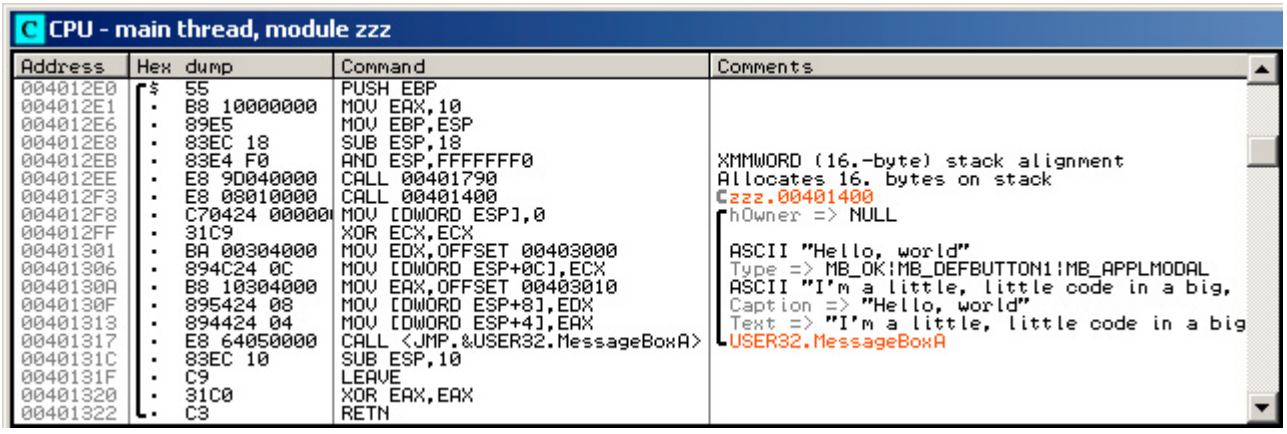
```
F: MOV EAX，[ARG.2]
PUSH ESI
MOV ESI，[ARG.1]
```

当然，分析器帮你做了这些东西.然而记录 ESP 会占据巨大的内存.如果内存很小，你可以关掉对系统 DLLs 的 ESP 追踪记录，而缺点是栈的轨迹将无法得到.

一些编译器不会将参数分开压入栈，相反的他们在栈上分配内存空间(SUB ESP，nnn)然后使用 ESP 作为索引来在新开辟的栈空间中写入参数.第一个双字参数是[ESP]第二个是[ESP+4]等等.

使用 MinGW(gcc)编译的样例程序:

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



注意:MessageBox()的顺序是(hOwner， Text， Caption， Type).MingW 已经改变了这个顺序，但是 OllyDbg 还是能够识别这些参数.

分支语句(switch)和级联式 IF 语句

为了实现一个 switch 语句，许多编译器将分支变量加载到一些寄存器中然后提取其中的一部分，就像下面这样:

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

这个代码序列包含了一级和二级的开关表，直接比较，优化以及其他事情.如果你足够深入 switch 的比较和跳转树中，你将会发现很难区分此 case 是何 case.然而 OllyDbg 会为你做这些事情.它会标记所有的 case，包括默认的，甚至尝试解释 case 所对应的跳转含义.比如'A'，WM_PAINt 或者 EXCEPTION_ACCESS_VIOLATION.如果命令序列并没有修改寄存器(比如仅仅是比较)，那么这可能不是一个 switch 分支，而是一个级联式 if 操作:

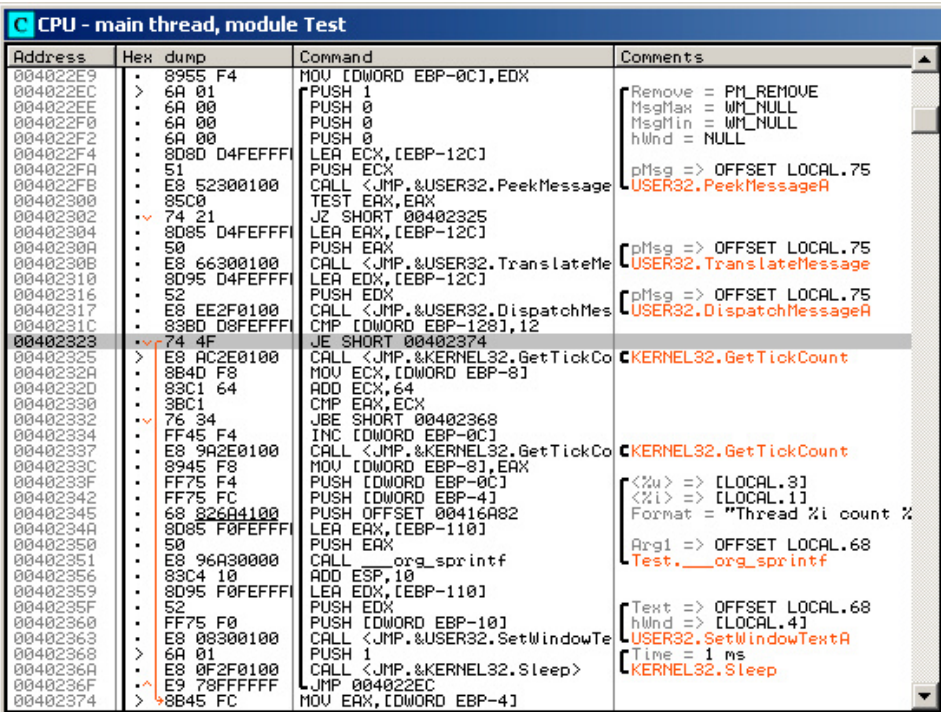
```
if (i==100) {...}
else if (i==101) {...}
else if (i==102) {...}
```

循环

循环由一组封闭式的连续序列命令组成，其中最后一个命令会跳到第一个命令处.循环一定是单个入口点以及没有限制数量的退出指令组成.循环对应与高级操作 do，while 和

for.OllyDbg 可以识别任何复杂的内嵌式循环.循环在汇编器中以圆括号标记.如果入口点不是循环里的第一个命令，那么 OllyDbg 将会将其用一个小三角标记.

下面是由测试程序(test.exe)所创建线程的主循环.所选择的命令是循环中的退出命令.长长的红色箭头表明了它的目的地址:

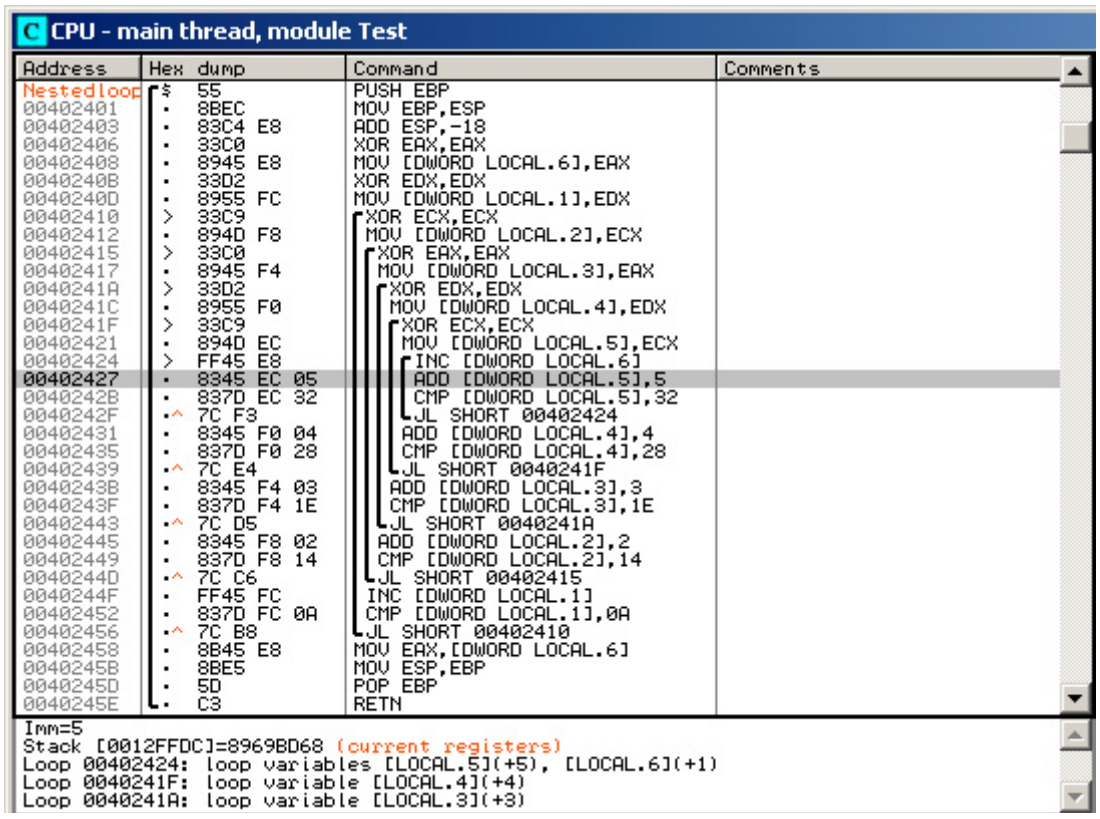


注意地址 00402351[10]处的 sprintf()函数调用，该函数是一个库函数，由编译器使用并且是未知的.但是代码中有其他的调用，因此分析器能够确定它的第二个参数是一个格式化字的字符串，并且已经将剩下的参数一一对应解码了.这是循环过程的 C 代码:

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

[10] 原版帮助手册中此处的地址为 00401AC1，系作者的错误——译者注

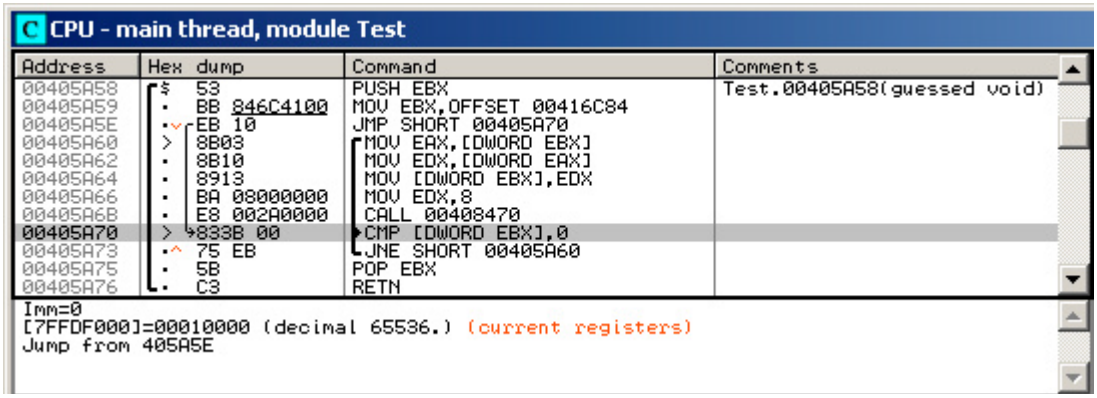
循环可以是嵌套的.Test.exe 中的过程 Nestedloops 是一个 5 层嵌套循环.




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

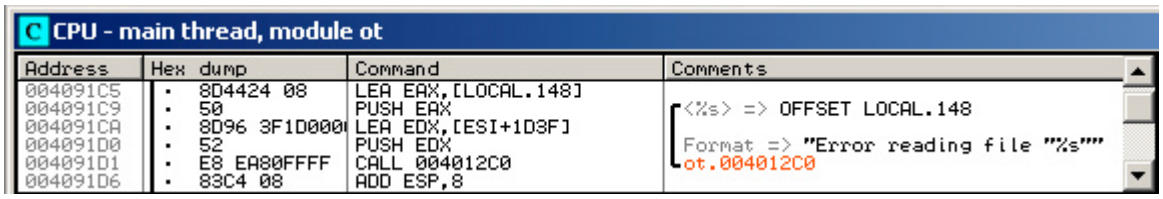
在信息窗口中与循环有关的注释可以做如下解释：起始于地址 00402424 每一次循环都有两个变量递增：变量 LOCAL5（本地栈上数据的第五个双字的数据）递增 5，变量 LOCAL6 递增 1 。它们分别对应源代码中的 m 和 n。位于 0040241f 地址变量 LOCAL4（l）递增 4 ，等等。

如果循环的入口不是框架中的第一个命令，那么会标记成一个小三角。



寄存器预测

看一下这个代码片段：



分析器意识到位于 004012c0 地址的程序，它的第一个参数是一个格式化的字符串。（事实上这个功能往往显示的都是错误的消息）这写程序可能有不同数量的 C 风格约定的参数，这表明程序必须在调用返回时删除这些参数。命令 ADD ESP，8，这条命令在调用之后就做到了这一点。它弹出了 8 个字节，或者两个双字。其中，第一个双字是格式化字符串的指针。因此，这个调用必须有一个附加参数。同样，格式化字符串也希望能有一个指向它的指针。

字符串的地址是该调用的第二个参数。它是以<%s> => OFFSET LOCAL.148 的形式被声明的。Ollydbg 以以下方式通知您：在此时的 PUSH EDX 的命令中，EDX 寄存器中储存的是本地变量 148 的地址，这个地址是由（LEA EAX，[LOCAL.148])命令加载到 EDX 中的。

符号==>在注释中的意思为“预测为等同于”例如 PUSH EAX 指令将格式串的地址压到栈中，004091CA 地址出的操作数与 0x1D37 相加，也等于 ESI 的值。但是 ESI 是在哪定义的？在之前的代码中调用代码的某个位置上。事实上，这个位置是在超过三百个字节之前。完整的过程是如下结构：

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

开始阶段，程序加载 ESI 静态数据块的地址。这个操作意义不大，但是它能使对线程的程序使用本地存储。PUSH ESI 或者 POP ESI 会使 ESI 的值免于修改，然后你就会看到 xxx 和 yyy 两个未知函数的调用。为什么分析器确定他们没有改变 ESI 的值？要么是直接从 xxx 和 yyy 的代码直接分析出来，要么是通过相应的分析告诉它的（**Advanced analysis | Unknown functions preserve registers EBX， EBP， ESI and EDI)**。所有的 Windows API 和许多单独编程的程序都是用 stdcall 调用方法进行函数调用。在 stdcall 协议中规定必须保护 EBX、EBP、ESI 和 EDI 寄存器。

已知的 API 函数

Ollydbg 提供了封装在以下函数库中的超过 2300 个标准 API 函数的说明：

·KERNEL32.DLL

·GDI32.DLL

·USER32.DLL

·NTDLL.DLL

·VERSION.DLL

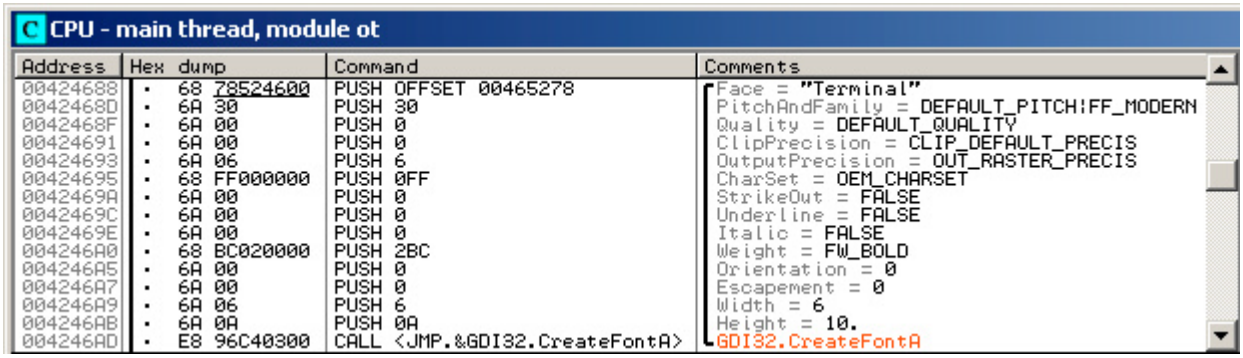
·ADVAPI32.DLL

·SHLWAPI.DLL

·COMDLG32.DLL

·MSVCRT.DLL

它也知道超过 10000 个符号常量，这些常量被分为 540 种，并且可以在代码中解码。



你也可以在汇编指令和算术表达式中使用这些已知常数。例如：CMP EAX，WM_PAINT 就是一个有效的汇编指令。

标准库函数

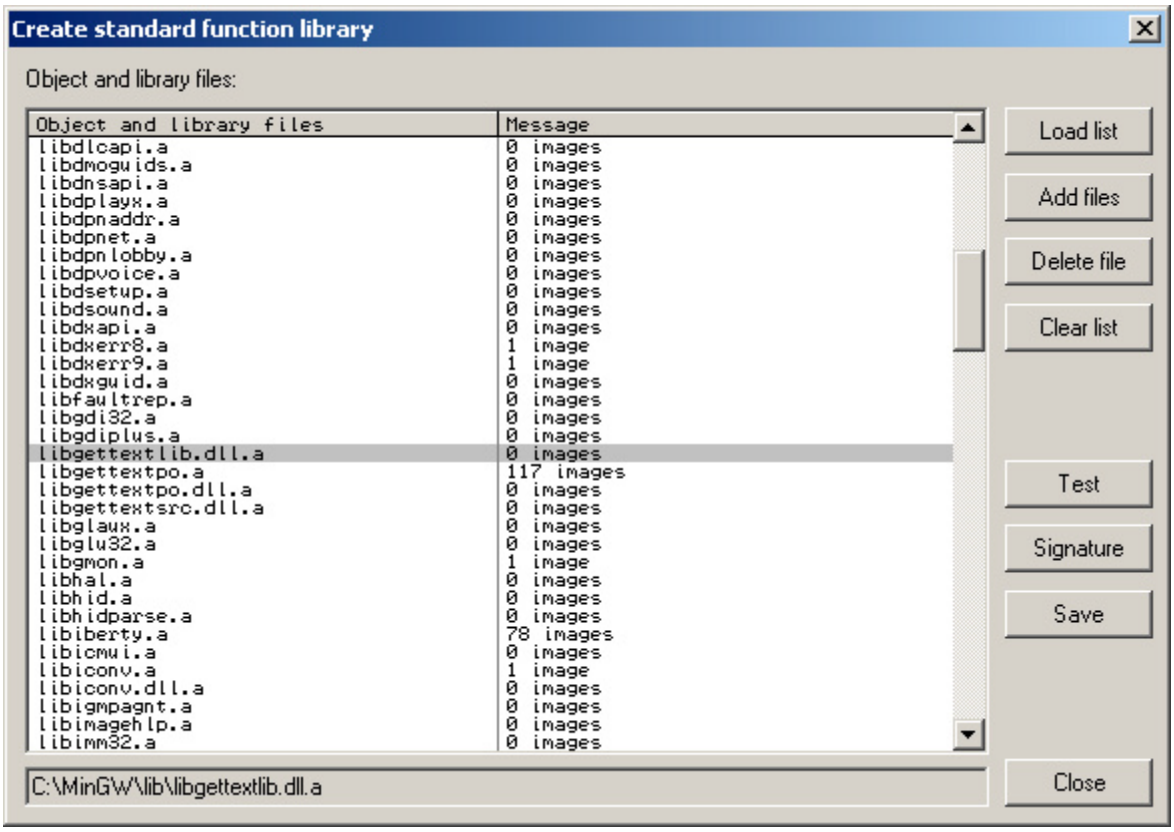
Ollydbg 可以像 Fopen、Sprintf 和 Malloc 那样，通过扫描函数库和通过比较比较库代码和调试程序的代码定位到标准库函数。这是通过两个步骤完成的。

协议许可规定，知识产权所有人可以阻止你使用除用给定编译器完成的代码链接以外的任何对标准库的操作。因此我们的例子是基于 MinGW 编译器去描述的。MinGW 是 GPL 发布的。

第一个步骤中，你必须选择编译器提供的标准库，并且创建 udl 文件。无论是作为库还是作为一个单一的对象文件，Ollydbg 都支持 OMF（Borland 公司的 Watcom）和 COFF（微软公司、GNU 等等）格式。通常它们有自己的扩展库和 obj。GNU 对对象文件使用.o，对库文件使用.a 。如果 MinGW 使用默认参数安装，那么主要的库在 C:\WinGw\Lib 这个目录下。

在 OllyDbg 主菜单上选择**调试|创建函数库**。点击**添加文件**，浏览目录 c:\MinGW\lib 并选择列表里全部文件和库。点击**打开**，它们将在对话框中变为红色，强调它们尚未被处理的事实。

现在按 **Test**（**测试按钮**），OllyDbg 读取文件并检查其内容。这个操作很快，且在第二列你会看到类似“48 images”的信息。这是库中的对象文件的数量包含二进制：



Libgettextlib.dll 不包含图像，怎么可能还比 1M 更大呢？此文件是一个导入库。选定后，其报告给连接器，比方说，_xmlTextWriteFlush 和许多其他函数是位于动态链接库 libgettextlib.dll。这些信息对于 OllyDbg 是没有用的。

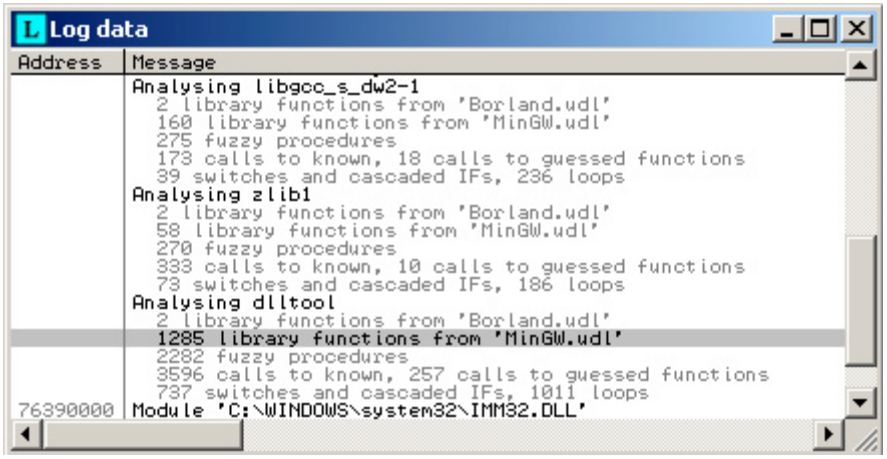
也有对象文件实现低级别功能，像位移或是长整型除法，它们位于 c:\MinGW\lib\gcc\mingw32\4.8.1(4.8.1 是 MinGW 的版本)。再按 **Add files（添加文件）**并选择它们。

现在你可以删除所有没有用处的图像文件。但这不会影响分析时间。因此：点击**保存**并选择子目录 udl(OllyDbg 默认查看 udl 库，可以在 **Options | Directories（选项|目录）**中更改)。

搜索标准库是比较快的，但取决于在库中图像的数量。在 udl 文件中它可以合理的不包括很少使用的库。

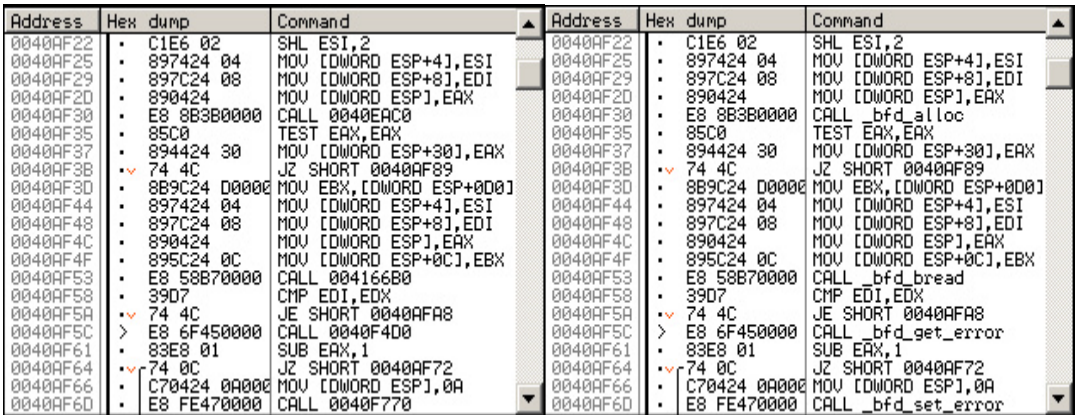
现在库已经准备好了。分析器将通过选项 **Advanced analysis | Detect standard library functions (*.udl).高级分析|检测搜索标准库函数(*.udl)**进行标准函数的搜索。

现在，我们加载一些文件使用 MinGW 库，例如 MinGW 有多种用途的 dlltool.exe。日志窗口通知我们分析器已经找到许多标准库函数。在它们的名字列表中被标记为“Analyser（分析器）”。但日志还报告了从 Borland 的库有两个函数，我们确定 MinGW 无法使用 Borland 的代码，难道不是么：



在 Borland 和 MinGW 中真的有一些非常简单的函数它们有相同的二进制代码。例如，它们可以从系统 DLL 调用标准函数。通常它们具有相同的功能。另一方面，标准库可以包含许多相同的函数（在某种意义上）不同的意思，像标准的析构函数。为了安全起见，OllyDbg 在 udl 库中不包含有歧义的函数。

在左面的屏幕截图是关闭搜索函数，右面的截图是开启：

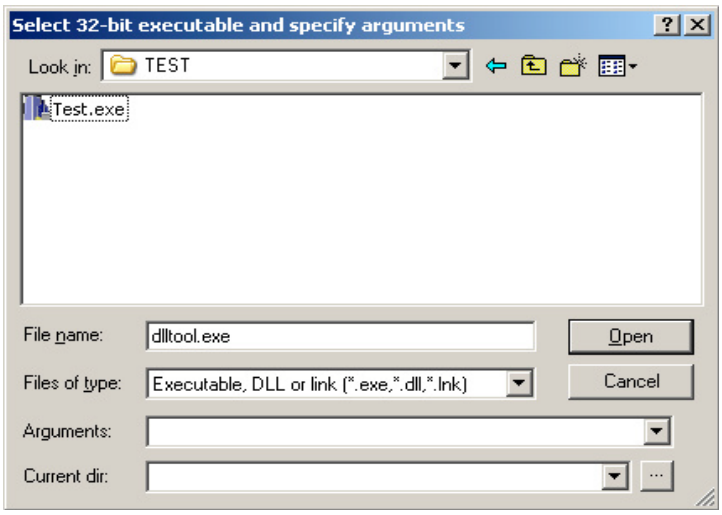


调试

打开程序

打开可执行文件的最简单方法就是将可执行文件拖拽到 OllyDbg 的主窗口。OllyDbg 支持可执行文件(.exe)和和链接库文件(.Ink)。单独调试 DLL 文件也是可以的，如下。

如果你需要确定命令行参数，从主菜单中选择“**文件|打开**”或者按 F3。然后在相应的行上选择你的参数。参数的长度一般限制在 1023 个字符之内。



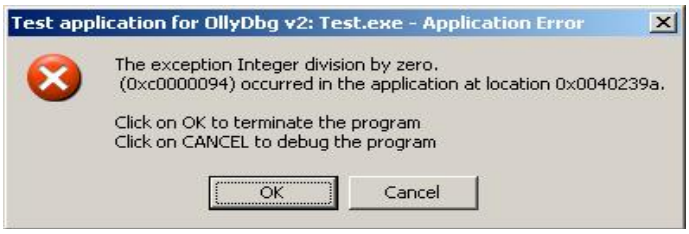
你可以在这里通过设置其中的 **Start | When starting application， make first pause at**（“**启动|当启动应用程序，产生第一暂停于：**”）告诉 OllyDbg 在哪里对新加载的应用程序进行第一次暂停，在下面选项中做第一次暂停：

- **系统断点**-应用程序将会中断在 ntdll.DbgBreakPoint。
- **TLS 回调**-在可执行文件中的第一个定义的 TLS 回调的入口处。如果有几个回调并且你想要检查他们，人为地设置断点。回调将会命名为<TLS_Callback_1>，<TLS_Callback_2>等等。如果没有回调函数，第一次暂停将会在可执行程序的主模块的入口点。
- **主模块的入口点**-在主模块的入口点，被命名为 PE 头。
- **WinMain**-如果 OllyDbg 知道这个入口，那么入口点的函数 WinMain()。如果 WinMain 的地址不知道，那么第一次暂停就会在主模块的入口点。
- **没有暂停**-一下都没有暂停，程序直接立即运行起来。

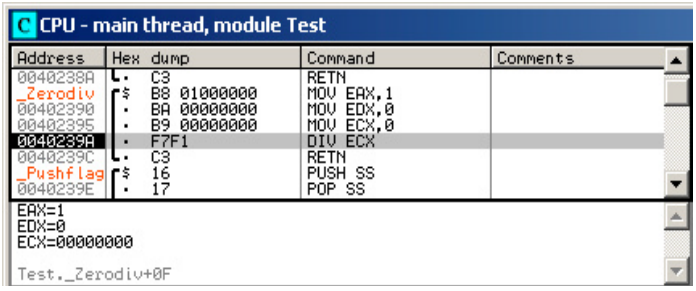
OllyDbg——一个即时性调试器

你可以将 OllyDbg 视为一个即时调试器。在选项对话框中打开 **Just-in-time**（**实时调试**）面板，然后按下 **Set Ollydbg**（设置 OllyDbg）。如果应用程序崩溃，你将会有更多的选项来调试它。（如果**你在附加程序前不检查确认**。所有有漏洞的程序将会自动的重定向到 OllyDbg-这可能不是你希望见到的）。

让我们来检查一下这种可能性。注册 OllyDbg，在脱机模式下打开 Test.exe 程序并按下按钮"0:0"



单击**"取消"**，然后 OllyDbg 会自动地弹出 CPU 窗口来指向导致异常的命令。



现在你可以像以前那样做任何事了。例如：你也可以用 EIP 指向下一个命令。0040239A 选择"RETN"。从弹出的窗口中**选择原始处**，然后在主菜单中执行 **文件|分离**（在 Windows NT/2000 下没作用）。如果没有事情发生，Test.exe 将会继续执行。

在分离时，OllyDbg 移除了所有断点。 它不会人为地继续被挂起的线程。

附加到正在运行的程序

假设你有足够的特权，那么你可以把 OllyDbg 附加到正在运行的程序上。从主菜单中，选择**文件|附加**..然后从正在运行进程的列表选择一个进程。

如果主线程被挂起（例如，应用程序创建时，标的状态是 CREATE_SUSPENDED）。OllyDbg 将会自动的继续运行它。因为不同的原因，在 Windows2000 下是不可能的。企图附加被挂起的程序最后也会崩溃。

附加到正在运行的进程是由 **Start | When starting application， make first pause at**（**启动|当启动应用程序，产生第一暂停于：**）这个选项控制的。

• **系统断点**：应用程序将会在临时的线程中暂停在系统断点处。当附加程序到调试器，Windows 在调试器的目录中创建了一个新的线程。这个线程将会执行 DbgBreakPoint()，这会让 OllyDbg 有机会做好必须的准备。这个线程在 OllyDbg 中被标记为暂时的。注意在 Windows2000 下，OllyDbg 是不能辨别线程是暂时的，进而认为它是一个普通的线程。

• **应用程序代码**-在主线程附加程序的时候的位置暂停程序

• **没有暂停**-应用程序应该尽可能地继续执行

调试子进程

OllyDbg 是一个单一进程的调试器。为了调试子进程，OllyDbg 会加载一个它自己的新实例，因此每一个子进程都会有一个它自己的 OllyDbg。这得在 Windows XP 或者更高版本的 Windows，而且只能是在父进程由 OllyDbg 开启的情况下才行得通。

由于 Windows 的限制，OllyDbg 不能调试孙子进程。这也就是说，如果你调试 A 进程。然后它派生出 B 进程，B 会传递给 OllyDbg。但如果 B 派生出 C，调试器将不会有任何注意，然后 C 就会自动运行。当然，你之后可以附加上 C 程序。

调试子进程的选项为”**Option|Debugging events|Debug child processes**”。

断点

OllyDbg 支持下面三种类型的断点：

• 在代码执行过程中设置的**软件断点**；

• 在内存入口处的**内存断点**，在内存中写入 and/or 代码；

• 在内存入口处的**硬件断点**，写在内存或者可执行代码处。

此外，你也可以在内存块的入口处设置断点（在 DOS-based Windows 的版本中不适用）这些在之前就讨论过。

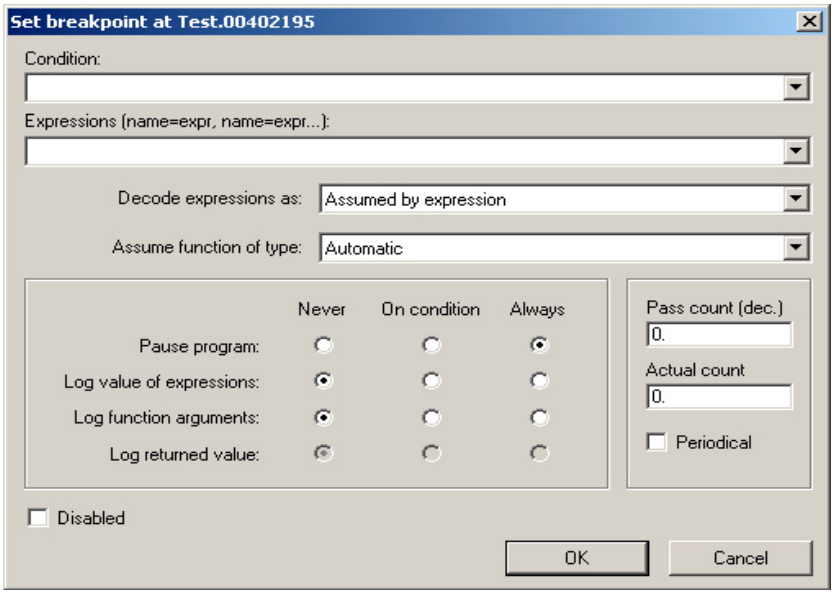
每一个类型都会有它自己的优点和缺点，具体优缺点的数据如下表所示：

类型	**软件断点**	**内存断点**	**硬件断点**
操作原则	第一个命令字节被 INT3 或者一字节的特权命令所代替（HLT，CLI，INSB 等）	入口保护使用的 VirtualProtectEx() 的改变	重新调用注册调试使用的 SetThreadContext()
数量	没有限制	没有限制	4
条件/日志	是	是	是
执行速度（如果没有碰撞）	没有影响	也许很低	没有影响
如果设置在错误的位置	在任何位置都可能崩溃	如果指向一段传给内核的保护内存，就会崩溃	跟以前一样的好
应用程序的检查	非常容易（读内存）	很容易（VirtualQuery）	容易（GetThreadContext()）

注意 Windows 只在 4096 字节的块中保护内存。因此，如果内存断点设置在经常变化入口变量的内存块中时，那么 OllyDbg 会得到很多错误断点，这些断点能大幅度降低运行速度。在一些情况中，沿着容许的命令模拟，也许会更快一些。至于细节，查看”Run trace and Profiling”。警告：**避免在栈中设置断点。**如果保护地址被传给系统，系统调用失败，就会终止进程。

每次命令执行或者内存被访问，简易的断点都会中断程序。只有在一些条件满足，且条件日志断点和一些额外的选项，你才会设置暂停程序的条件断点。例如：如果条件成立并且没有暂停程序，他们会协定一些参数的。

所有设置断点选项的对话框窗口都会有相似的结构，作为一个例子，下图是一个软件条件断点日志的对话框窗口。



记录断点支持四个动作：

• **暂停程序**-应用程序将暂停；

• **表达式的记录值**-在第二行中确定的表达式会被计算，然后将值协定传给日志窗口。你也许会使用多个用逗号分隔开的表达式，然后使用重复计数来显示连续的内存位置，看章节”Expressions”来寻求细节。在解码表达式中，可以通过指定表达式类型来影响解码控制。

• **记录函数参数**-如果断点被设置在所调用的已知的或者建议的函数上，或者如果你指定明确的函数（假设函数的类型），栈的参数将会解码，然后和日志窗口进行协定。

• **记录返回值**-如果断点被设置在立即调用函数的命令上，且这个函数返回一个值，寄存器 eax 中的内容将被解码并传输到日志窗口。

每一个动作，都会有下面三个选项：

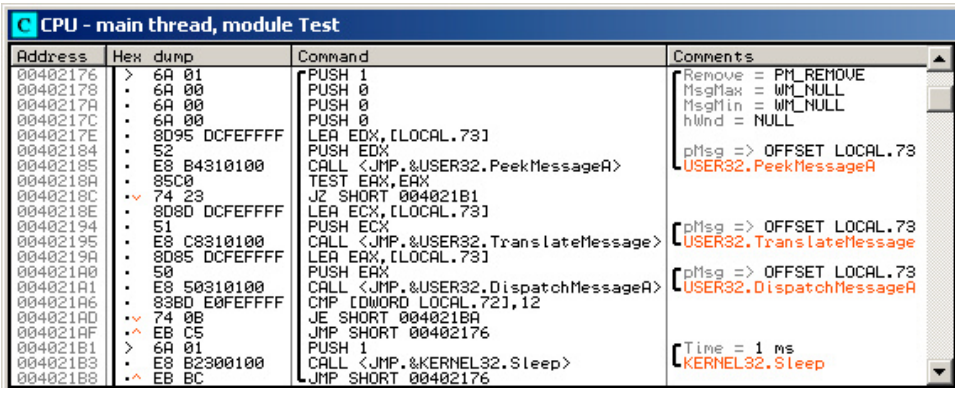
• **从不**-行为从未执行

• **有条件**-如果在第一行中的条件评价为 true，行为将会执行。

• **总是**-行为总会发生

此外，你也可以跳过几个通过确定传输数字的第一断点。如果定期检查的话，这些断点就能得到处理。

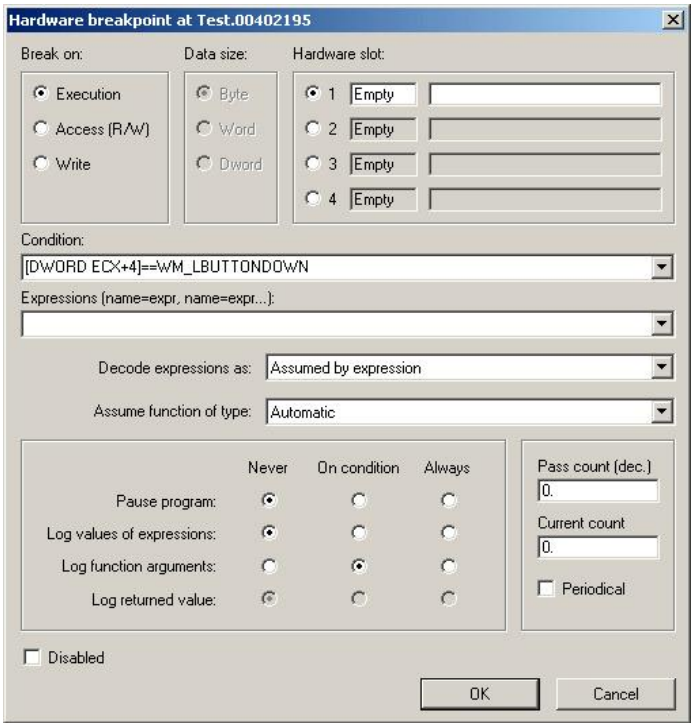
例如，假设我们想要协定 wm_lbuttondown 传递给 Test.exe 的所有消息类型，那么我们要使用的程序主要窗口如下：



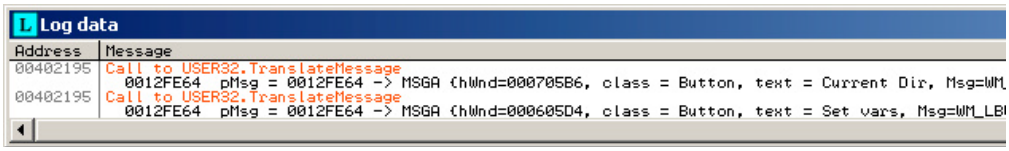
消息是由函数 PeekMessageA() 获取的。如果没有信息，那这个函数就会返回 0。程序 TEST EAX，EAX; JZ 004021B1 作为判断条件，如果消息丢失的话就会跳转。下一个函数，TranslateMessage()，用指针指向消息作为它自己唯一的参数。好的是，我们能够在函数上设置一个断点并记录下函数参数。然而，如果你这样做，OllyDbg 会协商传入所有的消息，而不仅仅是 WM_LBUTTONDOWN。

传递给 TranslateMessage()函数的结构体 MSGA 把消息代码作为它的第二个双字参数。注意 Test.exe 程序是一个 ASCII 应用程序，然后因此结构体 MSG 的 API 函数得到用来区别与 UNICODE 对应的 MSGW 的后缀 A。看一下反汇编：在调用的同时，寄存器 ECX 仍然保存指向这一结构的指针。结构体的指针是它第一个条目的地址。如果第一个条目是两个字（4 字节），那么第二个条目的地址就应该是 ECX+4。这个条目都在内存中。为了获得它的目录，我们必须获得整个框架的内存地址并且确定该块内存的大小。（由于两个字是默认的，内存大小也是可选的）因此正确的表达式是[DWORD ECX+4]== WM_LBUTTONDOWN。常量 WM_LBUTTONDOWN 是已知的，我们不需要来找到头文件来获取它的值。

让我们来使用硬件断点。选择一个调用命令，右击然后选择 **Breakpoint | Hardware log...** （**断点|硬件记录**...）硬件断点也许在不是命令执行的内存入口处或者写入内存（但是不在它们的连接处）选择**执行**的命令执行处触发，在这种情况下数据大小总是 1 个字节并且面板总是灰色的。现在选择 4 个可用断点中的一个。在我们的情况下，所有的断点都是空闲的，按照下面方式使用，它们也没什么。



确定你的选择，运行程序然后按下按键**"Current dir"**和**"Set vars"**。现在看一下日志窗口：

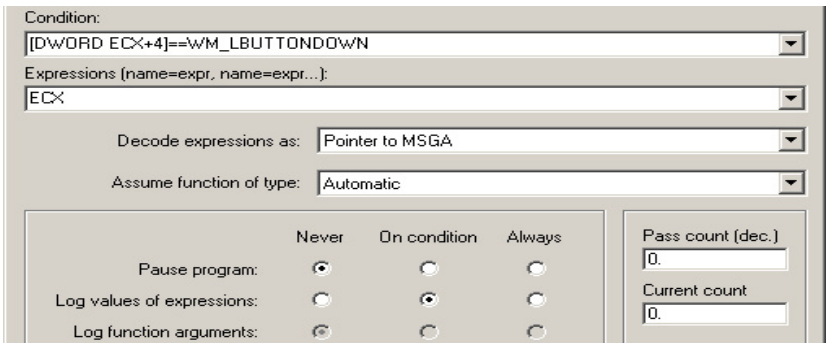


看一下第一入口。TranslateMessage()的参数是两个字：0012FE64。这个参数叫做 pMsg，这个参数是个指向结构体 MSGA 的一个指针。结构体 MSGA 的第一个元素是 hWind--包含文本文字"Current dir"的 Button 键的窗口句柄。现在来看第二个 MSGA 成员，和消息 WM_LBUTTONDOWN 相同的消息来标示 Msg。

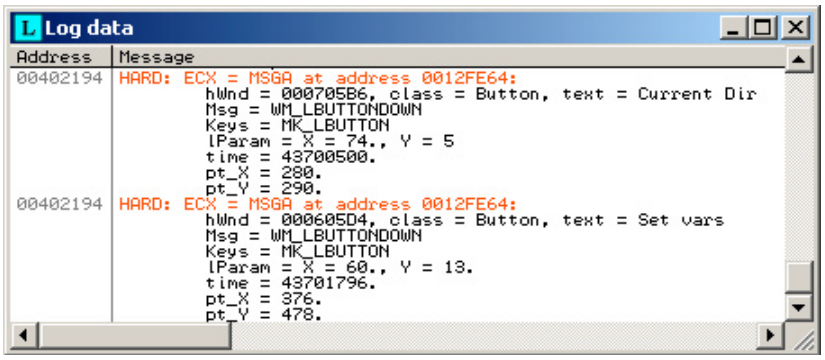
在这个例子中，我们很幸运，因为 TranslateMessage()是一个很有名的函数（在 API 中很具有代表性）。那如果不是这种情况我们该怎么做呢？让我们试试简洁不同的方法，看一下下面的程序：

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

第一个指令把指向 MSGA 类型结构体的指针加载到 ECX 中。移除之前的断点，然后在 PUSH ECX 处设置断点。现在我们必须确定传输的表达式：ECX。这个表达式必定被解释为 MSGA 的指针。最后，表达式的值只有在和之前一样的情况时才会记录下来：



再按同样的按钮并且检查日志。现在下面是更加具有可读性的：



运行跟踪和分析

运行跟踪是一种通过命令执行和来传输调试程序的命令的方法。在这个方法中，我们可以定位最经常执行的代码部分，侦查跳到哪儿或者仅仅只是回溯程序到执行在一些事件发生之前的地方。

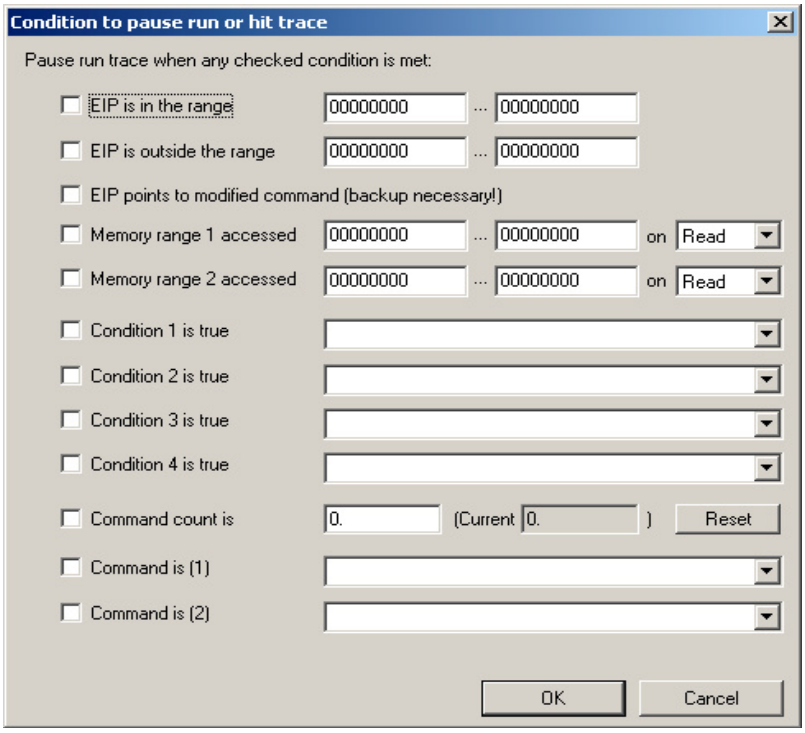
简而言之，运行跟踪在下一个命令设置断点然后继续执行。然而这个方法是很慢的。Windows 需要 10 微秒来暂停程序并把调试事件传递给调试器。这使得 Windows XP 中，运行跟踪被限制在每秒最多 10000-30000 条命令，而在 Win7 中也许每秒 30000~70000 条命令。

为了加快运行跟踪的速度，OllyDbg 可以不用将调试指令传递给调试器而模拟内部命令的执行。这个模拟器目前限制到 55 个常用的命令，比如 MOV，ADD，CALL 或者条件跳转指令。如果命令是未知的或者无法模拟（像 SYSENTER），OllyDbg 就把它传递给应用程序。尽管如此，执行程序速度能达到每秒 300000~600000，简单指令甚至能达到每秒 1000000 条命令。在很多情况下，这对于“几乎 real-time”Windows 应用程序是足够的。

每一个跟踪命令都传输到大的循环缓冲区中。这约定包括地址，内部寄存器和符号的目录。如果你需要，你也可以储存命令，FPU 寄存器和可访问内存的目录。注意每一个选项都

会要求更多的内存并且减少与缓冲区相匹配命令的数量。如果缓冲区是 256MB 并且额外的所有内存都关闭了，它也可以保存上至 16，700，000 条命令，而其他的-只能保存 7，000，000~10，000，000 条。

大多数运行跟踪选项也都应用于命中跟踪。大概最有用的跟踪特征就是能够在一些事情发生时暂停程序的能力了。（从主菜单中选择 **Trace|Set condition**…）

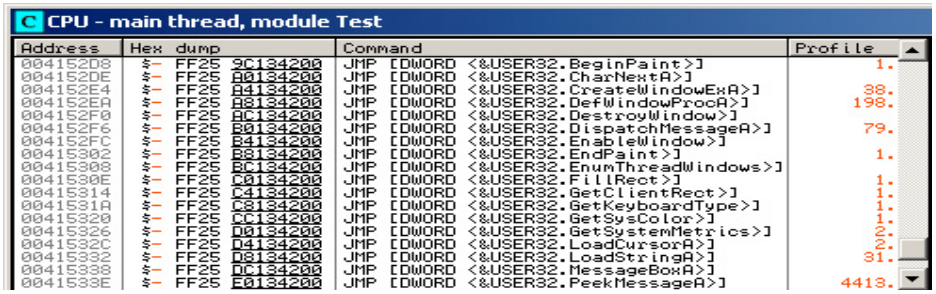


大多数的选项都是自描述性的。选项 **EIP points to modified command** （**EIP 指向被修改的命令**）可以被用来查找程序的入口点。（顺便说一下，在这个情况下的中断跟踪更快一些）。当你开始跟踪并且上文中的选项是激活的，OllyDbg 会将实际的命令代码和备份的代码比较，当出现不同时，程序暂停。当然，备份的必须存在。最简单的方法是让 **Debugging | Auto backup user code**（**调试|自动备份用户代码**）选项处于永久激活状态。

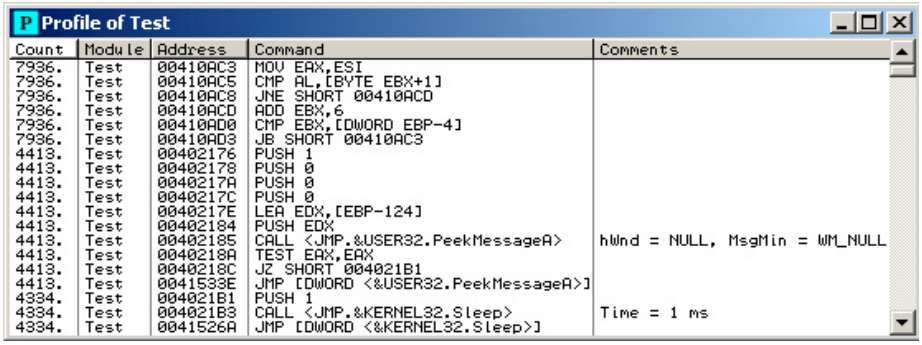
在内存入口处暂停的范围可以通过设置内存断点来实现。然而，80x86CPU 只可以保护 4096 字节的内存块。如果使用内存块的内存断点设置为活跃，执行程序会导致大量错误调试事件发生。为了修复这个问题，OllyDbg 必须移除内存保护，执行单个导致异常的命令并且再一次存储内存断点。这要求大量的时间。如果快速命令模拟器是活跃的，运行跟踪就有意义——它能够快于内存断点高达 20 倍。

任何有效的可以模拟成 TRUE(非 0)或者 FALSE(0)的表达式都是一种情况。举个例子来说，EAX==0 或者([BYTE 450002]&0x80)！=0。寄存器被实际的线程所掌控。条件表达式的判定很快并且对于运行跟踪速度只有很小的影响。

如果你需要知道每一个跟踪命令的执行频率，在反汇编器中选择 **Comments | Show profile** （**注释|展示轮廓**），或者 **Profile selected module**（**描绘已选择的模块**），再或者在运行跟踪窗口中的 **Global profile**（**全局轮廓**）。在第一种情况中，注释行中会显示这条命令在跟踪数据中出现了多少次：



单独的描绘窗口列举出了被跟踪的命令，通过它们的频率来进行分类：



正如你所看到的，在地址 00410AC3 处的一小段代码执行地最为频繁。它位于在启动时自动执行的常规初始化区域。正如我们预想的那样，第二段最为频繁的代码，是属于 Windows 循环语句的主线程中。命令 PUSH 1 压入参数来消除栈中的 API 函数 PeekMessageA()。

运行跟踪遵从下面说明：

- **调试|允许快命令模拟器**-允许 OllyDbg 模拟一些经常在 CPU 内部使用的命令（在调试器中的目录），因此加速调试；
- **运行跟踪|运行跟踪缓冲区的大小**-分配内存给带有运行跟踪数据的循环缓冲区。头一个约定，最大命令数保持在 30000-60000 条命令；

• **运行跟踪|不能进入系统 DLLs**-要求 OllyDbg 在跟踪模式中立即执行调用 Windows API 函数。注意如果 API 函数调用用户空间的回调函数，它们也将不会被跟踪。

• **运行跟踪|总是跟踪字符串命令**-要求 OllyDbg 跟踪字符命令，像 REP MOVSB。如果这个选项没被激活，每一个 MOVSB 迭代都将会分割传递开来。

• **运行跟踪|记住命令**-在跟踪缓冲区中存储跟踪命令的副本。只在调试应用程序使用自我修改代码时需要。

• **运行跟踪|内存记录**-存储标好跟踪缓冲区中内存地址的目录

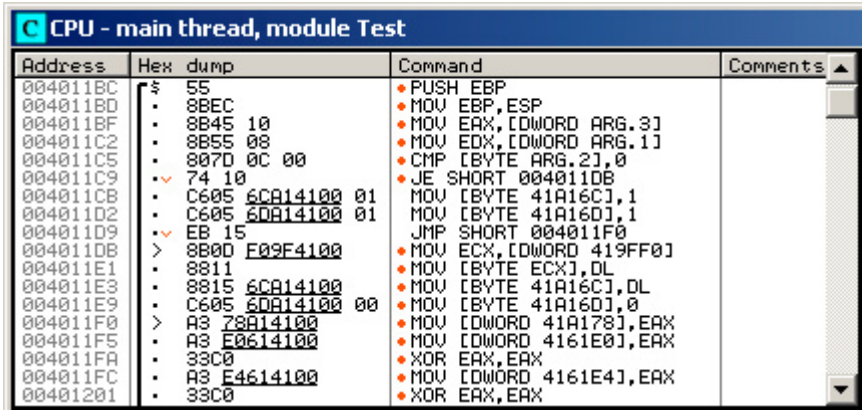
• **运行跟踪|记住 FPU 寄存器**-存储浮点-指向跟踪缓冲区的寄存器

• **运行跟踪|同步 CPU 和运行跟踪**-改变 CPU 选择然后每一次改变在运行跟踪协议中的选择就更新 CPU 寄存器

命中跟踪(Hit trace)

调试的意义在于尽可能的发现并移除被调试程序的 Bug。为了达到这个目的，你需要执行程序的每一个子例程和分支。如果不这样做，那么可能的错误就会保留在程序中。但是你如何知道某一段代码是否被执行了？命中跟踪将会给出答案。

命中跟踪从实际的 EIP 开始。OllyDbg 在当前还没有跟踪过的所有分支处下软断点，在遇到跟踪断点之后，命中跟踪移除该断点并将命令标记以命中。通过这样的措施，我们可以检查特定的分支代码是否被执行过。下图中的示例表明：00411CB 处的代码到现在还没有命中。



命中跟踪执行非常迅速.在短暂的启动周期结束后，应用程序几乎以实际执行时的速度运行.在非直接分支和调用处，比如 CALL [0x405000] 或者 JMP [0x123456+EAX*4]处就可能发生一些问题.这种情况下 OllyDbg 提供两种选择:如果**命中跟踪(Hit Trace)|每次都检查目的地(Check destination each time)**启用，那么 OllyDbg 将会在那些命令处维持断点。这虽然花费

时间，但是保证了安全性。当**命中跟踪(Hit trace)|分析数据猜测目的地(Use analysis data to guess destinations)**选项启用，那么 OllyDbg 将会假设所做分析将能够正确的确定所有可能的目的地并对所确定的目的地进行标记。第二种方式要快的多，但是可能导致遗漏掉一些分支，或者如果条件为 false 的分支指向数据或者命令中部时将会发生崩溃。如果你犹豫不决，那么就用第一个选项吧。

一些反病毒程序将代码段放在内核内存中(通常在地址 0x80000000 及其以上处)。OllyDbg 不能在内核内存断点处设置软断点，因此将会继续以单步执行模式运行到用户内存处。

其他的命中跟踪选项如下:

Hit trace | Set breakpoints on known callbacks （命中跟踪|在一直的回调函数上下断点）- 如果激活该选项，OllyDbg 在命中跟踪启动后，将会在所有已知的回调函数处设置跟踪断点。这可以使得跟踪来自 Windows API 的函数调用比如 SendMessage()。

Hit trace | When next destination is analysed as data:Continue hit trace / Pause hit trace（**命中跟踪|**） -当分析器错误百出的将无效代码识别为数据时，或者如果被调试程序是自我修改的或动态创建代码时，可以启用这个选项。对于第二种情形需要注意的是，INT3 断点可能对执行造成致命影响。

Hit trace | When next destination is outside the code section:

如果被调试程序动态创建代码或者动态的从磁盘加载代码，设置 INT3 断点可能会导致崩溃。单步追踪是最安全的，然而也是最慢的方式。

Hit trace | Keep trace between sessions –

命中跟踪将会保存到.udd 文件中并且在你重启程序的时候恢复。

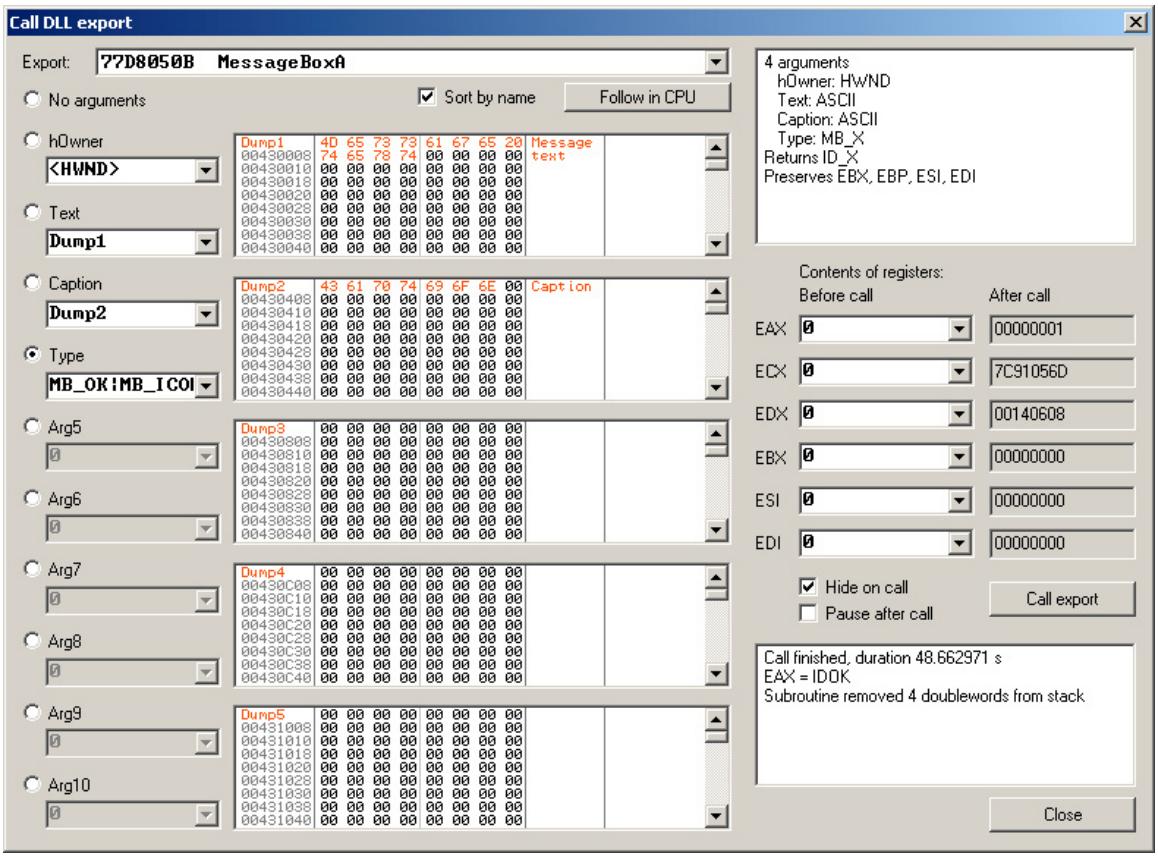
直接调试 DLL

Loaddll.exe

OllyDbg 可以调试单独的动态链接库(DLLs)。Windows 不能够直接的启动 DLL，因此 OllyDbg 使用一个叫做 loaddll.exe 的小型可执行程序。这个程序以打包的资源形式寄宿于 Ollydbg.exe 中。如果你尝试打开的文件是一个动态链接库，那么 OllyDbg 将会自动的提取 loaddll.exe 并启动它，然后将库名作为参数传入。

Loaddll.exe 本身只是一个没有任何控制选项的窗口。它只是加载 DLL 并无限的重复窗口循环，等待来自调试器的命令。调试界面在 OllyDbg 中实现。

我会在 MessageBoxA(USER32.DLL 中的 Windowsde API 函数)例子里介绍这个功能。在库加载后，你可以像其他任何应用程序一样设置断点，应用补丁。当一切准备就绪，从主菜单 **Debug | Call DLL export** ，弹出的窗口如下:



在顶部是由库导出的所有函数列表.选择 MessageBoxA。这个 API 处于内部的数据库。OllyDbg 知道 MessageBoxA 需要 4 个参数，甚至能够告诉你它们的名字和类型。第一个参数是父窗口的句柄，第二个和第三个是指向 NULL 结尾的 ASCII 字符串，最后一个参数是变

量名和几位数的组合 MB_xxx。

由于参数的个数是已知的，调试器自动勾选左侧代表 4 个参数的单选按钮。你可以进入对应的编辑控制区域或者从下拉列表中选择预定义的参数。

<HWND>-loaddll.exe 所拥有的窗口句柄

<HINST>-loadll.exe 实例

Dump1..Dump 10-指向 10 个预定义的内存区域，每一个区域为 1024 字节。虽然 5 个内存 Dump 显示在对话框中的 **Dump1** 到 **Dump5** 区域中，但是你可以在鼠标右键的弹出菜单中选择 **Go to|Expression（转向|表达式）**进行修改。

如果函数要求寄存器参数，那么可以在右侧的控制区域输入。标准的 Windows API 函数从来不用寄存器作为参数。

现在我们来设置参数。第一个是父句柄，这里<HWND>是最好的选择。第二个参数虽然是一个指向显示在消息框中的 ASCII 字符串指针，我选择的是 **Dump1** 区域，然而，任何其他处于通信区域的的内存空间地址也都可以。我们想要显示”Message text”文本。选择 Dump1 中的一些字节，然后从鼠标右键弹出菜单中选择 **Edit|Binary edit（编辑|二进制编辑）**并在 ASCII 字段域中输入”Message text”。

同样的方式，选择第三个参数(**Dump2**:”Caption”)。设置第四个参数为 MB_OK|MB_ICONERROR.OllyDbg 知道这些常量。

准备完成后，按下 **Call export**，消息框将会出现:

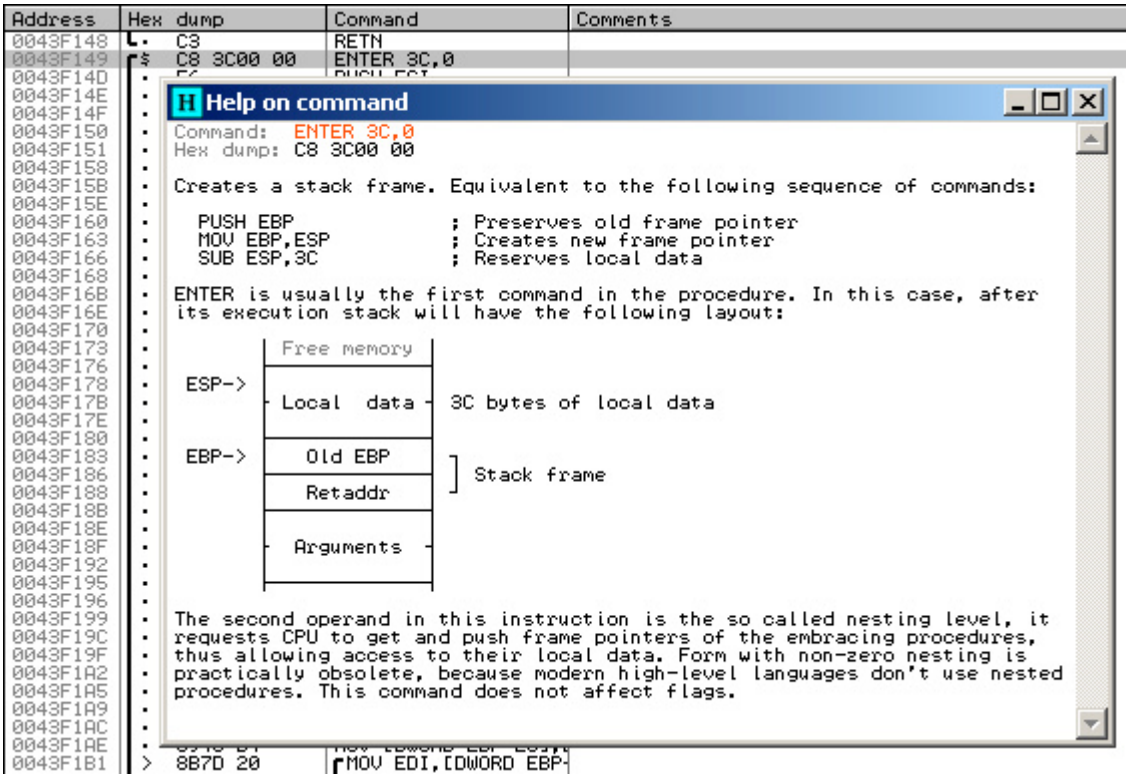
如果勾选了选项 **Hide on call**，调用 DLL 窗口将会临时消失.按下消息框中 OK 按钮，窗口将会回到原来的地方。注意右下角:”EAX=IDOK”。它指明了 MessageBoxA()返回了 IDOK(常量 0x1)。另一个消息，”sububroutine removed4 doublewords from the stack" 可以确认这是 Pascal 分割的函数，它使用 4 个栈类型的参数。

Loaddll.exe 使用汇编写成，源码可以自由获取。有好几个补丁区域你可以给程序添加其他功能。注意如果 Loaddll.exe 已经在 OllyDbg 目录，那么 OllyDbg 将会使用已经存在的 Loaddll.exe。

帮助

命令帮助

OllyDbg 集成了 80x86 命令的帮助文档。选择命令并鼠标右键选择 **Help on command**(快捷键 **Shift+F1**)选项:



OllyDbg 会尝试描述所选择的命令以及它的操作。当前的帮助文档能够提供所有的整型数据，FPU 以及系统命令的帮助。MMX，3D 和 SSE 命令在以后将会加入。

API 函数的帮助

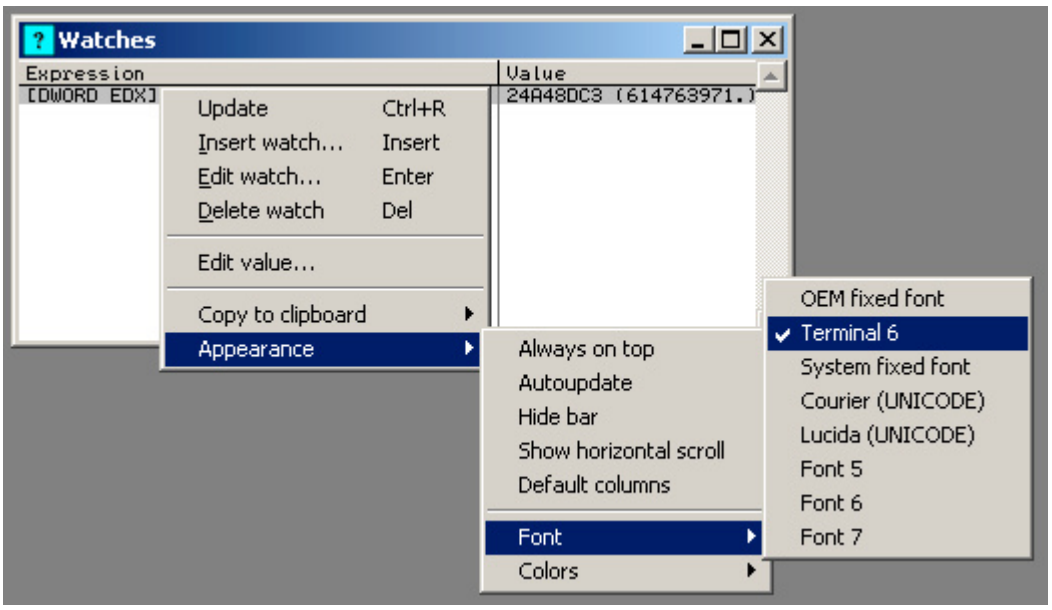
如果你有 Windows 的 API 帮助文件，并且是.hlp 或者.chm 格式的(比如，以前很好用的 win32.hlp)，那么你能通过 **Options | Directories | Location of API help file** 将其添加到 OllyDbg 当中去。为了在 API 处获取帮助，选择 API 函数调用或者 API 的入口点，然后按下 **Ctrl+F1**(或者从鼠标右键弹出菜单选择 **Help on API function**)。

由于法律等因素，帮助文件没有包含在 OllyDbg 的发行版中。

自定义

字体

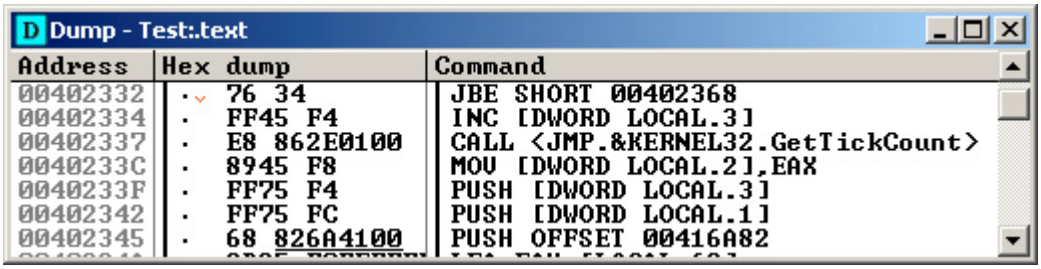
OllyDbg 的每一个窗口都有 8 种字体供选择:



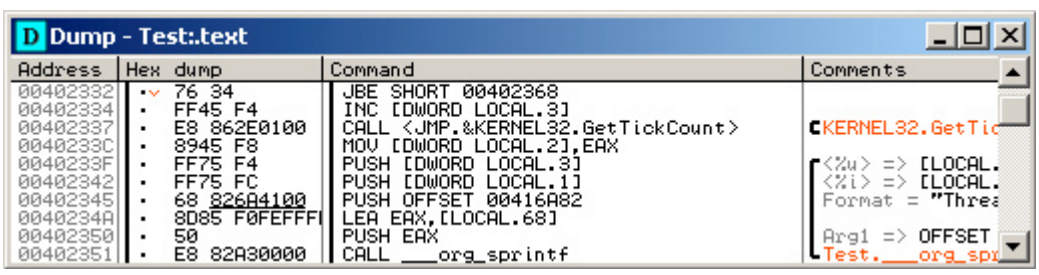
默认情况下，OllyDbg 使用”Terminal 6”字体.这种字体较小而且可读性很强，以使 OllyDbg 能够显示足够的信息。但是 Terminal 6 只有基本的 OEM 字符集而且字体太小，可能不适合与视障人士。

你可以选择不同的预定义字体或者重新定义已经存在的字体。下面这些字体是你第一次启动 OllyDbg 的时候是以选择的:

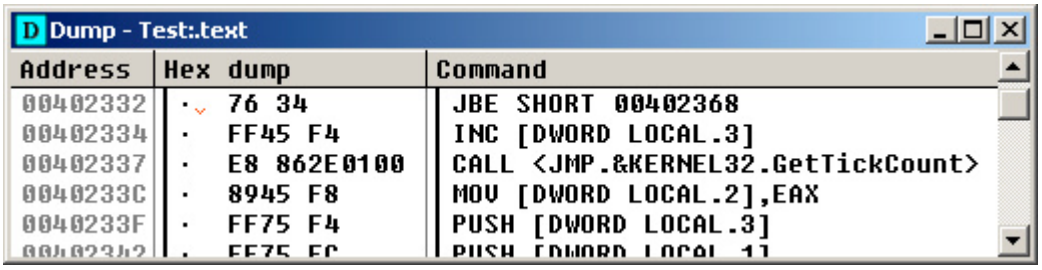
OEM fixed font:



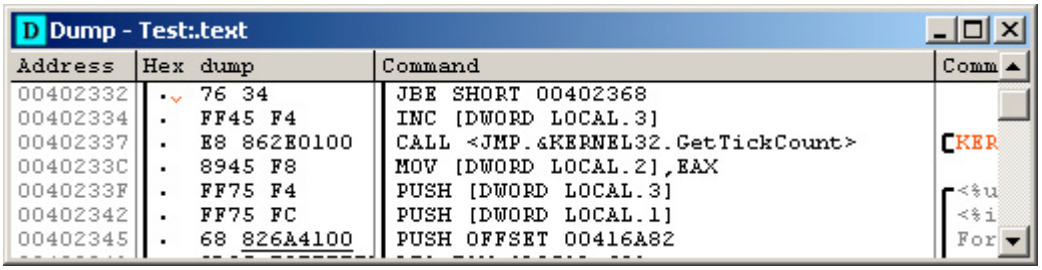
Terminal 6:



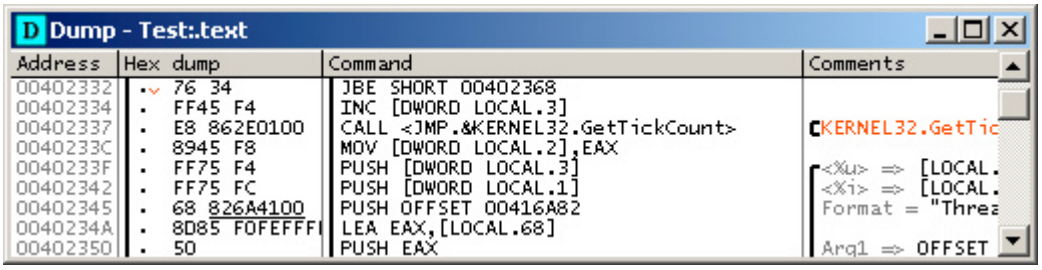
System fixed font:



Courier (UNICODE):



Lucida (UNICODE):



字体 5，6，7 初始化分别是和 Terminal，System，Courier 一样。注意你的 Windows 操作系统上可能缺失一些字体(由其他字体替换)。

OllyDbg 只允许尺寸固定的字体，也就是说所有的字符都有相同的宽度。因此你能使用 Times New Roman 字体-该字体就是不适合。一些字符，比如 W 被切掉了一部分，另一个比如 I，也会被调整的很不适合显示。

注意固定字体并不是有人以为的字体宽度固定。Kanji 和 Hangul 符号通常是 ASCII 子集的两倍宽.如果你想要在内存 Dump 中无剪切的显示它们，我建议你打开选项 **Dump | Use wide characters in UNICODE & multibyte Dumps**。

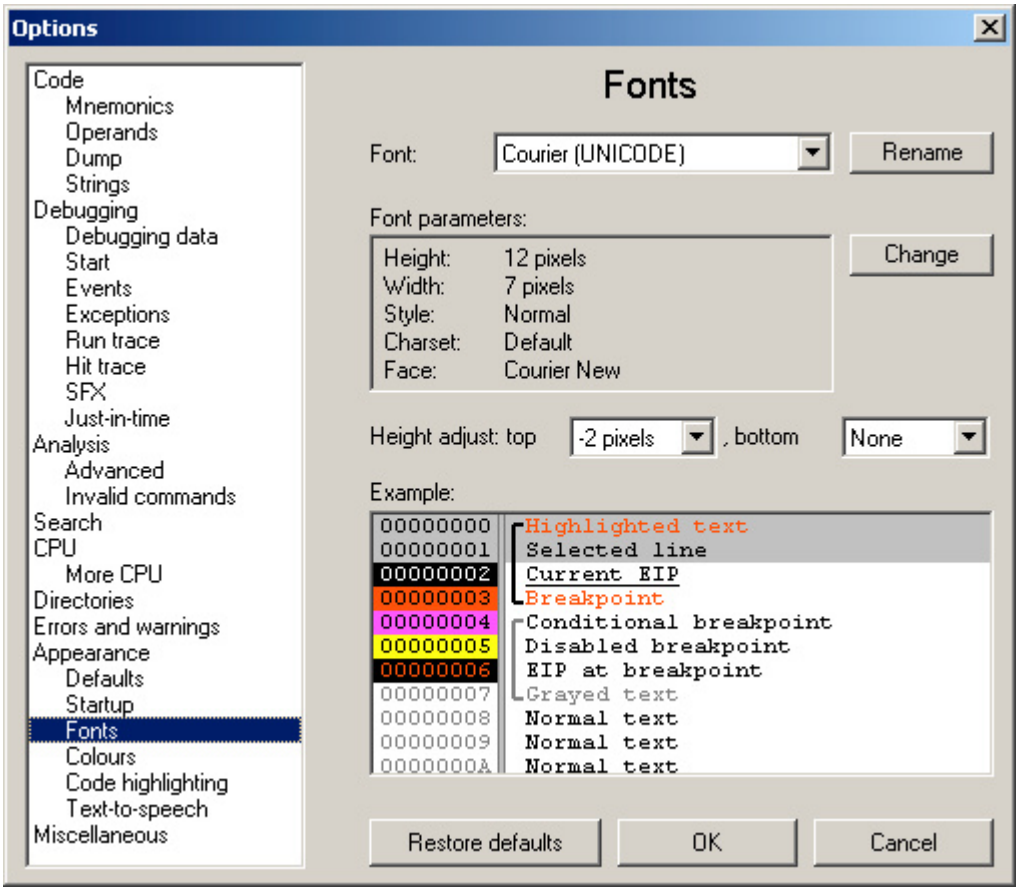
如果你想改变字体的外观，打开 **Fonts** 选项面板。请注意:这里的修改只对当前选择的字体生效。如果反汇编器使用 Terminal 6 而你编辑的是 OEM 固定字体，你会发现反汇编面板中的字体没有发生任何改变。要么调整 Terminal 6 字体，要么在反汇编器中选择 OEM 固定字体。注意任何的修改都是即时生效的，比如当你在标准字体对话框中按下 **Ok** 或者 **Apply**

按钮时。但是如果你从顶部 **Fonts** 选项面板中选择不同的字体，这并不会自动改变 OllyDbg 窗口中选择的字体!

回到 **Fonts** 选项.按钮 **Rename** 可以帮你给当前选择的字体重命名。这个动作是没有什么实质用处的。因为实际上，OllyDbg 窗口中的字体是由槽索引选择而不是由字体名称来选择。如果反汇编器中选择的是 Terminal 6 字体而你将 Terminal 6 字体重命名为 Monitor 10-4，那么反汇编器中一点变化也没有，只有外观菜单显示当前选择的字体是 Monitor 10-4。

按钮 **Change** 调用字体标准对话框，允许你选择字体以及定义字体尺寸和外观.它列出了电脑上安装的所有固定宽度字体。

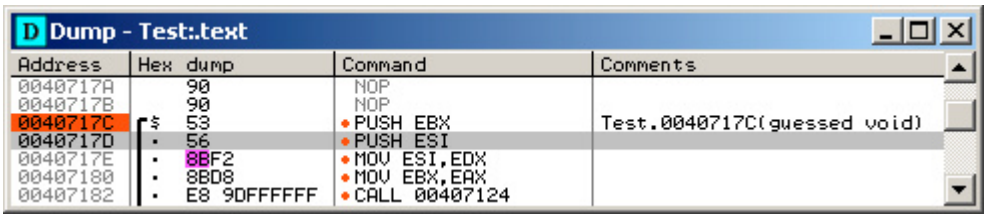
通过控制 **Height adjust top** 和 **Height adjust bottom** 来添加或移除显示行上方或下方至多 5 像素的空隙。比如，Terminal 6 中的大写字母是 7 像素高而字体高度(定义在字体描述中)是 8 像素。屏幕上线与线之间间隔 1 像素是不够的，特别是当固定字体使用下划线时。为此，OllyDbg 额外在每一行的顶部都添加了一像素。然而另一方面，Courier New 却太大了，因此有必要从顶部移除 2 像素。



着色

有八中颜色主题决定窗口表格中使用的着色方式。每一种主题都可以被应用到任何一中窗口表格中去(鼠标右键弹出菜单 **Appearance | Colours**).只预定义了前四种主题：

Black on white（白底黑字）:

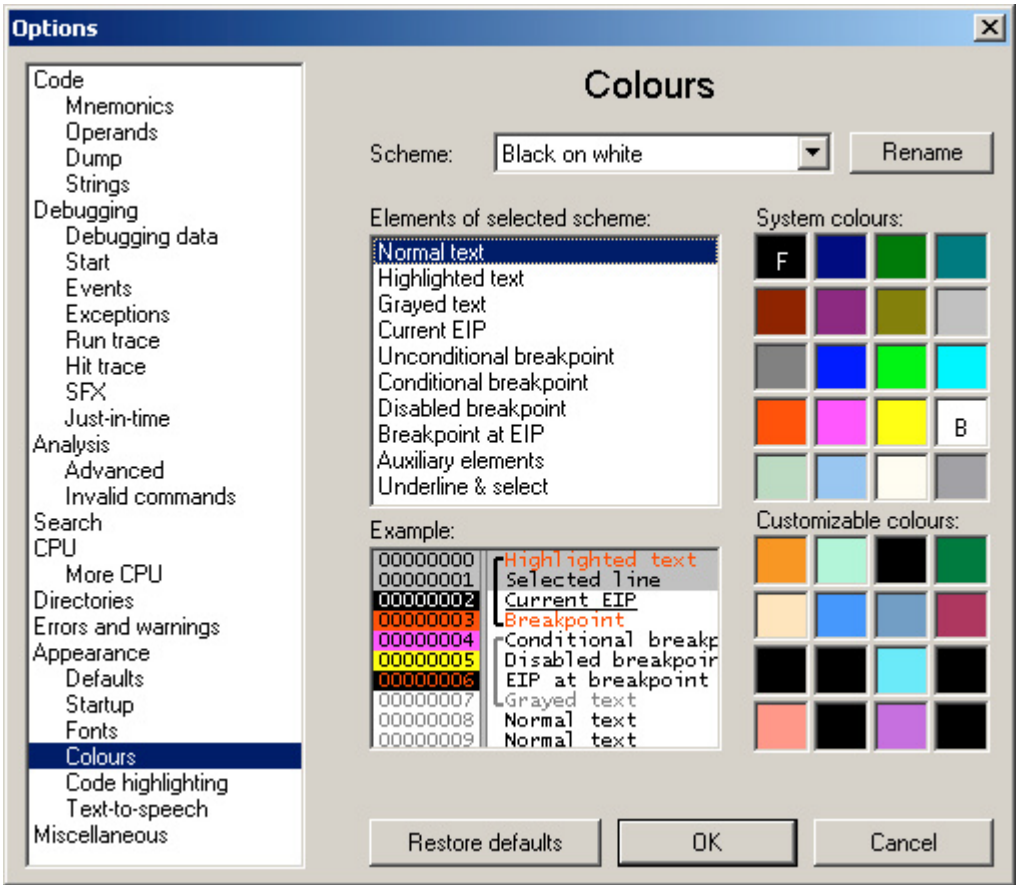


Yellow on blue(蓝底黄字):

Marine（海军蓝）:

Mostly black(几乎全黑):

剩下的四个默认配色方案，假设命名为方案 4-到方案 7，重复前四个方案。配色方案可以自由编辑。



和字体完全一样，你在这里所做的改变只适用于在调色盘顶部选中的方案。如果在反汇编器中选中的方案是 **Black on white（白底黑字）**，而你去编辑 **Mostly black（几乎全黑）**，你会看到在反汇编器里没有任何改变。

元素或多或少都有描述性的名称，辅助元素现在已经不使用了（但在插件中可能还使用）。

OllyDbg 定义了 20 种基本的颜色，展示在调色盘的上半部分。它们属于 8 位调色盘的默认颜色。在使用古老的显卡的情况下，应减少调色盘的变化次数。另外你可以定义 16 中定制的颜色——只需双击相应的块。

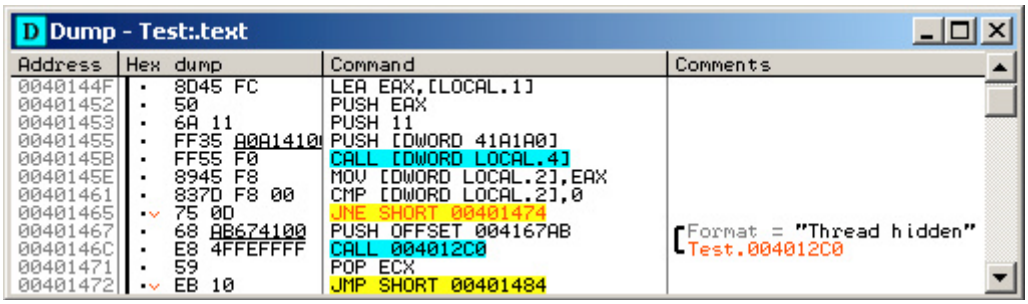
要改变前景的颜色，只需选中元素，然后用鼠标左键点击想要的颜色即可。字母 F 移动到上面，表明是前景。鼠标右键选择背景颜色，变化会立即生效。如果可视的列表窗口中的颜色方案和你编辑的恰好一样，你可以马上看见结果。**Restore Default（恢复默认设置）**按钮会将颜色调会预定的方案。**Cancel（取消）**按钮会撤销所有的修改。

Code Highlighting（代码高亮）

Ollydbg 提供 4 七种可行的代码高亮方案，但最初 OllyDbg 只提供三种：

Christmas tree， definitely exaggerated（**圣诞树**，当然是夸张的说法）：

Jumps and calls（跳转和调用）：



Memory access（内存访问）：

可突出的对象是指令和它们的操作数。指令被分成下面几个组：

无条件的跳转	JMP， JMP FAR
有条件的跳转	JE， JNZ， ...; JCXZ
PUSH/POP	PUSH， POP， PUSHF， POPF（PUSHA 和 POPA 被解读为纯指令）

调用	CALL， CALL FAR， INT
返回	RET， RETF， IRET
FPU，MMX，SSE	所有的 FPU，MMX，3DNOW!，SSE 和 AVX 指令
错误，系统和特权指令	I/O：IN， OUT， INS， OUTS， ...；系统指令：ARPL， SYSENTER，...；特权指令：HLT，INVLPG， LGDT，...；错误指令：LEA ESI，EDI。注意 UD2 是一个有记录的纯指令。
填充	NOPs被用作程序之间填充：NOP， LEA ESI，[ESI]，…
修改指令	和备份不同的指令
纯指令	所有剩下的指令，例如 MOV 或 ADD

如果选项 **Highlight operands（突出操作数）**被激活，操作数就会单独地突出：

通用寄存器	EAX， ESI， AX， AL， ...
FPU，MMX，SSE 寄存器	ST(0)， MM0， XMM0， YMM0， ...
选择器和系统寄存器	FS， CR0， DR0， ...
堆内存	包括 ESP 和 EBP 的内存地址
其他内存	不包括 ESP 和 EBP 的内存地址
指向内存的常量	一些模块的指向内存的常量
其他常量	所有剩下的其他常量

和代码突出紧密相关的是寄存器突出。如果 Dump 窗口按反汇编显示，打开右键菜单选择 **Highlight register|<general-purpose register>（寄存器突出|<通用寄存器>）**。这个选项的优先级比突出代码和可见的指定寄存器及其组件高。例如你选择 EAX，OllyDbg 会突出 EAX， AX， AH 和 AL。

Shortcuts（快捷键）

默认的快捷键是基于 Borland 工具使用的方案（我是 Borland 的大粉丝）。你们大部分可能对 Microsoft's Visual Studio 更熟悉一些。并不是问题——快捷键可以快速地编辑。从主菜单中调用 **Options | Edit shortcuts...（选项|编辑快捷键…）**，选中项目然后选择按键组合。如果这个组合不被允许、保留或与已存在的快捷键冲突，你会收到对应的警报。如果有冲突但是你还是选择 **Apply（应用）**，发生冲突的快捷键会被删除。

对现有设置的修改被保存在 *ollydbg.ini 中*。按 **Save** 按钮，你可以保存你的快捷键到独立的文件中。你可以自由地分配修改——实际上，我鼓励你去做这个。

GUI Ianguage（GUI 语言）

OllyDbg 2.01 在用户界面中支持多种语言。注意这仍是一个处于实验状态的功能。

下面的内容是你如何将 OllyDbg 翻译成新的语言的方法。下载 *ollydbg.lng* 文件，这是一个 UNICODE 文本文件：

```
EN English

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

// ANALYSER:227
EN Analysis interrupted

...
```

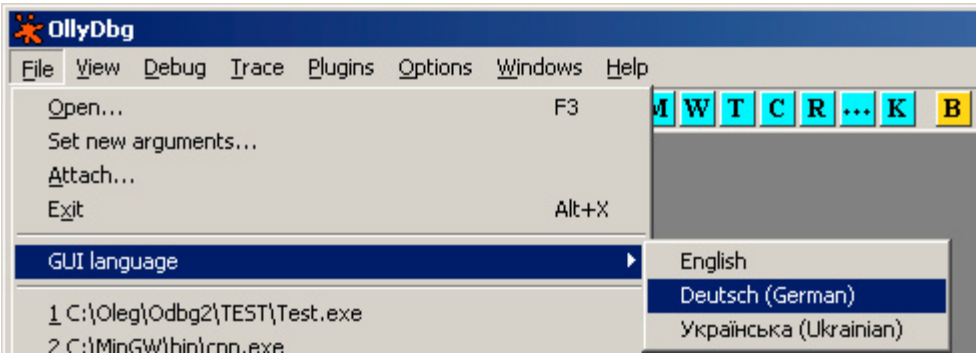
//ANALYSER:206 等行是表明 OllyDbg 来源中第一个出现的字符串的注释。你可以从文件中去掉它们，但是它们减少了故障排除时麻烦。

首先你必须定义一个或多个新语言，分配两个字母给它们作为标识符，然后直接在”EN English”这行之后加上标识符，后面再跟上语言的名字，先用这门语言然后用英文。

然后一个一个的翻译 OllyDbg 使用的全部 4100 个文本字符串。每个被翻译的行必须以这个语言的标识符作和最初的一行作为开始。在翻译格式说明符的时候请特别注意，它们的顺序必须保持不变。不要用看起来相似但是有不同编码的字符来代替格式字符——这可能会导致崩溃！菜单选项中的‘&’符号的位置必须被改变，以便定义新的菜单快捷键。‘$’（美元）符号有特殊的含义，让它保持在相似的位置上。

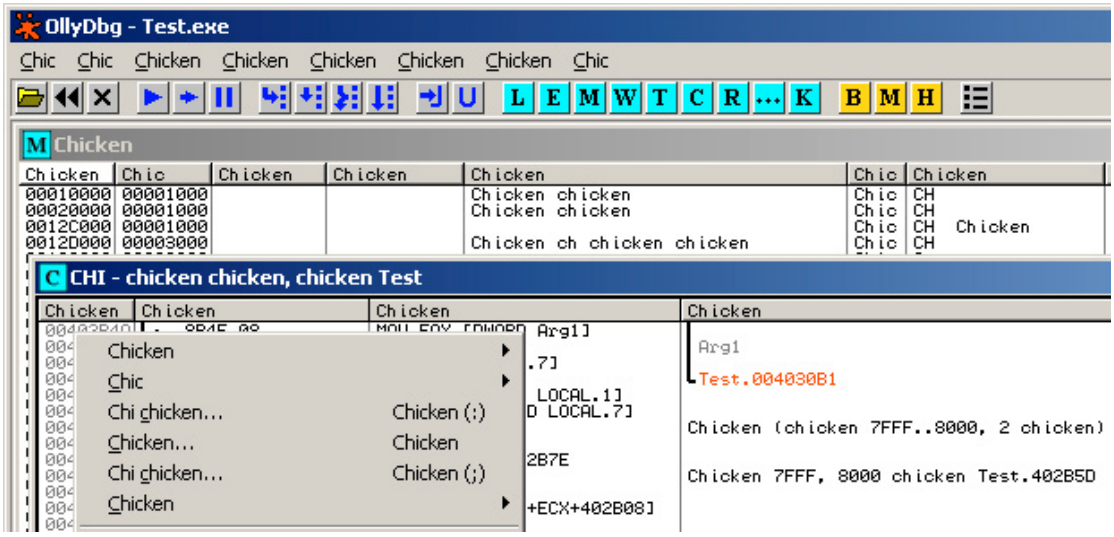
```
EN English
DE Deutsch (German)
UA Українська (Ukrainian)

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

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

...
```

当你准备好了，保存这个文件到包含 *ollydbg.exe* 的目录下，然后启动 OllyDbg。要改变语言，打开主菜单然后从 **File|GUI language（文件|GUI 语言）**中选择新的语言。这个菜单一直是英文的。



语言会立即改变，并不需要重启调试器。只有少数元素，例如窗口标题，会在你重新打开窗口的时候先被翻译。解析器创建的可翻译的尝试被保存在.*udd* 文件中，要改变它们，需要重新进行解析。

要调试这个功能，我已经把所有的字符串翻译成了 chicken。下面的链接会提供一个很好的解释说明：www.youtube.com/watch?v=yL_-1d9OSdk，PDF 文档在这里：isotropic.org/papers/chicken.pdf 。你可以从 OllyDbg 的主页上下载这个 chicken 语言文件。


OllyDbg - Test.exe
Chic Chic Chicken Chicken Chicken Chicken Chicken Chic
L E M W T C R ... K B M H
Chicken
Chicken Chic Chicken Chicken Chicken Chic Chicken
00010000 00001000 Chicken chicken Chic CH
00020000 00001000 Chicken chicken Chic CH
0012C000 00001000 Chic CH Chicken
0012D000 00003000 Chicken ch chicken chicken Chic CH
CHI - chicken chicken, chicken Test
Chicken Chicken Chicken Chicken
Chicken
Chic
Chi chicken... Chicken (;)
Chicken... Chicken
Chi chicken... Chicken (;)
Chicken
Arg1
Test.004030B1
Chicken (chicken 7FFF..8000, 2 chicken)
Chicken 7FFF, 8000 chicken Test.402B5D

给 Ollydbg 重命名

如果你想给 Ollydbg 的可执行文件重命名，比如说，alias.exe，它会自动更改一些设置：主窗口的名字将变为 alias，初始化文件将改名为 alias.ini 窗口的名字将变为 alias_ODWin，alias_ODMDI 等等。但是，有一个问题：插件都是静态连接到 Ollydbg.exe 的，它们不能连接到其他名字的可执行文件上。

为了解决这个问题，Ollydbg 试图在主模块在内存上的名字也做暂时性的修改。这种机制在 Windows XP 系统上运行良好，但是我不能保证在其他 Windows 版本的系统也能这样。

致歉

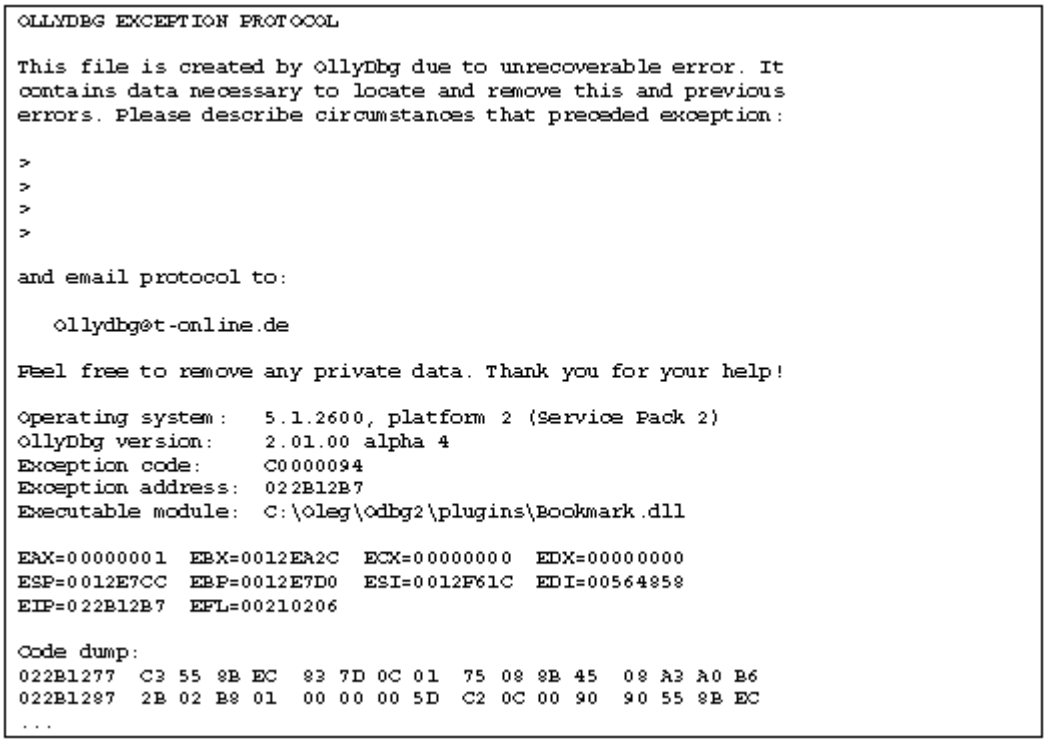
我们为给您带来的不便深表歉意

道格拉斯 亚当斯

英语并不是我的母语，请原谅我所有的语法错误，如果您能为我错误的语法和词汇进行纠错，我会十分高兴的！

我也要为 C 语言的语法错误向大家道歉，这些错误通常产生一些处理器不喜欢的命令和数据而引起窗口报告。当然，这全都归咎于我，很多时候我都辞不达意，否则，以编译器的出色能力，是一定会具备执行力的。所以，请原谅它，并将错误日志报告发送给我，我将纠正这些错误。

下面的图片是错误日志的样例：



```
OLLYDBG EXCEPTION PROTOCOL

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

>
>
>
>

and email protocol to:

   ollydbg@t-online.de

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

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

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

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

为您造成了不便，我深表歉意，如果您错过了一些有用的功能，请联系我，但是不要抱太大期望~

所有的内容就是这样了！

——*Oleh Yuschuk，　也可以称作 Olly*