



This document is provided for informational purposes only and Microsoft makes no warranties, either express or implied, in this document. Information in this document, including URL and other Internet Web site references, is subject to change without notice. The entire risk of the use or the results of the use of this document remains with the user. Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 1999-2002 Microsoft Corporation. All rights reserved.

Microsoft, Microsoft Press, MSN, MS-DOS, Windows, Windows NT, CodeView, MSDN, Visual C++, Visual Studio, Win32, and Win64 are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Intel is a registered trademark of Intel Corporation. Itanium is a registered trademark of Intel Corporation.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

TABLE OF CONTENTS

MODULE 1: INTRODUCTION TO DEBUGGING	9
Module 1 Overview	11
Introduction	12
General Terms	13
General Terms - Continued	14
General Terms – Continued	15
General Terms – Continued	16
Debugging Terms	17
Debugging Terms - continued	18
Debugging Terms – Continued	19
Types of Debuggers	20
MODULE 2: DEBUGGER INSTALLATION & SETUP	23
Module 2 Overview	25
Debugger Installation	26
Symbol Files	27
Symbol Files: Overview	27
Proper Symbol File Directories	29
Installing Windows Symbol Files	31
Symbol Servers and Symbol Stores	33
Using SymSrv	34
Using SymStore	37
Module 2 Labs	38
MODULE 3: INTRODUCTION TO DEBUGGER OPERATION	39
Module 3 Overview	41
Opening a Crash Dump File	42
The Debugger Command Window	43
Setting Paths and Loading Files	45
Checking for Symbol Problems	48
Ending the Debugging Session	51
Displaying Memory	52
Displaying Registers and Flags	54
The Call Stack	55
Stacks	55
Adding Items	56
Removing Items	57
Thread Stacks	58
Nested Function Calls	60
Viewing the Call Stack	63

Module 3 Labs	65
MODULE 4: KEY CONCEPTS & DATA STRUCTURES 67	
Module 4 Overview	69
Key Concepts and Data Structures	70
Component Overview	70
Processes	71
Kernel-mode Components	72
Debugger Commands: !drivers	74
Processes and Threads	76
Debugger Commands: !process	77
Debugger Commands: !thread	79
Process ID (PID) Displayed by TList	80
EPROCESS	81
KPROCESS	83
Process Environment Block - PEB	84
Debugger Commands: !processfields and DT	85
ETHREAD	88
KTHREAD	89
Thread Environment Block - TEB	91
Debugger Commands: !threadfields and DT	92
Thread Scheduling	93
Thread States	95
Voluntary Switching	96
Preemption	97
Quantum End	98
Priority Boost and Decay	100
Debugger Commands: !thread, !pcr, !ready, !locks	101
Trap Dispatching	105
Interrupt Dispatching	106
Interrupt Types and Priorities)	107
Predefined IRQL's	109
Interrupt Levels vs. Thread Priority Levels	111
Exception Dispatching	112
Debugger Commands: .trap, !pcr, !idt	114
System Service Dispatching	118
Memory Management	119
Address Space Layout	120
System Address Space Layout	121
Page Fault Handling	123
System Memory Pools	124
Debugger Commands: !vm, !memusage, !pte, !pfn, !dd, !pool	125
I/O System Components	128
I/O System Structure and Model	130
Device Drivers	132
Structure of a Driver	133
I/O Data Structures	135
File Object	136
Driver Objects and Device Objects	138
Driver Object	140
Device Object	141
Processing an I/O Request	143
I/O Request Packets	144
Debugger Commands: !thread, !irp, !irpfind, !devobj, !drvobj	146
Module 4 Labs	147

MODULE 5: CRASH DUMP ANALYSIS I	149
Module 5 Overview	151
Kernel-Mode Dump Files	152
Kernel-Mode Error Handling	152
Varieties of Kernel-Mode Dump Files	153
Creating a Kernel-Mode Dump File	156
Analyzing a Kernel-Mode Dump File	159
Opening a Kernel-Mode Dump File	159
Collecting Information from a Kernel-Mode Dump File	164
Module 5 Labs	166
MODULE 6: KERNEL DEBUGGING I	167
Module 6 Overview	169
Setting Up a Kernel Debugging Session	170
Configuring Hardware for Kernel-Mode Debugging	170
Cables Used for Debugging	172
Testing the Connection	173
Configuring the Target Computer	175
Configuring the Target Computer (continued)	176
Configuring Software on the Host Computer	180
The Kernel Debugging Session	181
Beginning the Debugging Session	181
Breaking Into the Target Computer	183
Examining the Target Computer's State	185
Synchronizing with the Target Computer	186
Ending the Debugging Session	187
Module 6 Labs	188
MODULE 7: UNDERSTANDING DISASSEMBLED CODE	189
Module 7 Overview	191
x86 Architecture	192
CPU Architecture	192
Intel CPU Registers	194
Register Types	195
x86 Instructions	197
Operands	197
Memory Reference Operands	199
Accessing Memory	201
Arithmetic Instructions	203
Logic Instructions	204
String Instructions	205
Stack Instructions	206
Thread Stacks	207
Pushing Items onto the Stack	208
Popping Items Off the Stack	209
Pushing Items Using a Stack Pointer	210

Flow Control Instructions	211
CALL and RET Instructions	213
Annotated x86 Disassembly	214
Relating C to Assembly	214
Module 7 Labs	218
MODULE 8: CALL STACKS	219
Module 8 Overview	221
Calling Conventions	222
Understanding Calling Conventions	222
Calling Convention Example	223
Prologue and Epilogue Code	224
Typical Prologue Code	225
Stack Layout Before Running Prologue Code	226
Stack Layout After Prologue Code	227
Typical Epilogue Code	228
The STDCALL Calling Convention	229
STDCALL Example	230
The CDECL Calling Convention	232
CDECL Example	233
The FASTCALL Calling Convention	235
FASTCALL Example	236
Analyzing the Stack	238
Interpreting the Stack	238
Displaying Call Stacks	239
Return Addresses (Saved EIP)	241
Frame Pointer Linked List	243
Using the Frame Pointer (EBP)	244
Frame Pointer Omission	246
Frame Pointer Omission (FPO)	246
FPO Frame Type Information	247
Observations of FPO Functions	248
Displaying a “Lost Stack”	249
Module 8 Labs	252
MODULE 9: CRASH DUMP ANALYSIS II	253
Module 9 Overview	255
Bug Checks (Blue Screens)	256
Bug Checks	256
Commands for Analyzing Bug Checks	258
Two Broad Categories of Bug Checks	260
Failed Assertion Bug Checks	261
Exception Bug Checks	262
Exception Bug Checks (Continued)	263
Trap Dispatching	264
Exception Dispatching	265
Changing Contexts	267

The Debugger’s “Contexts”	267
Setting the Register Context	270
Setting the Register Context (continued)	271
Setting the Register Context (continued)	272
Additional Debugger Features	273
Types of Workspaces	274
Information in Workspaces	276
Keeping a Log File	278
Debugging in Assembly Mode	279
Debugging in Source Mode	281
Using Debugger Extensions	284
Module 9 Labs	286
MODULE 10: KERNEL DEBUGGING II	287
Module 10 Overview	289
Live Debugging	290
Controlling the Target	290
Using Breakpoints	293
Reading and Writing Memory	295
Reading and Writing Registers and Flags	298
Remote Debugging	299
Remote Debugging Through the Debugger	300
Remote Debugging Through Remote.exe	303
Other Debugging Techniques	306
Debugging a Stalled System	307
Debugging Timeouts	309
Debugging Memory Leaks	313
Using Driver Verifier	316
Driver Verifier Options	320
Driver Verifier Options – Continued	324
Module 10 Labs	332
MODULE 11: USER-MODE DEBUGGING	333
Module 11 Overview	335
Configuring and Starting the Debugger	336
Configuring Software for User-Mode Debugging	336
Starting the Debugger	337
Using TList to get a Process ID (PID)	338
Attaching to a Running Process	340
Spawning a New Process	341
Basic Debugger Operations	342
The Debugger Command Window	342
Choosing Processes and Threads	343
Controlling Exceptions and Events	344
Debugging an Application Failure	351
Ending the Debugging Session	352

User-Mode Dump Files	353
Creating a User-Mode Dump File	353
Analyzing a User-Mode Dump File	357
Debugging Memory Leaks	360
Debugging User-Mode System Processes	366
Redirecting NTSD Output to KD	366
Debugging CSRSS with NTSD	367
Debugging WinLogon with NTSD	369
Module 11 Labs	370

Module 1: Introduction to Debugging

Module 1 Overview

- **Introduction**
 - What is debugging
- **Commonly Used Terms**
 - General Terms
 - Debugging Terms
- **Type of Debuggers**
 - Character-based console debuggers
 - Microsoft Windows-based debugger

What You Will Learn

After completing this lesson, you will be able to:

Understand what we mean when we say debugging.

Understand the basic terms used in this workshop.

Describe the different debuggers available from Microsoft®.

Source

Debugging Tools for Windows® Documentation

Introduction

- **Debugging**
 - To find and remove errors (*bugs*) from a program or design.
- **Debugging (As defined for this workshop)**
 - Using debugging and other tools to isolate the most probable cause of a system failure or application error.

Debugging

To find and remove errors (bugs) from a program or design.

Debugging (As defined for this workshop)

Using debugging and other tools to isolate the most probable cause of a system failure or application error.

General Terms

- **Process**
 - A process is a container for a set of resources used by the threads that execute the instance of the program
- **Thread**
 - A *thread* is the entity within a process that Windows schedules for execution
- **Call Stack**
 - A fundamental data structure used to keep track of function calls and the parameters passed into these functions
- **Register**
 - A register is a very fast temporary storage location in the CPU

Process

Although programs and processes appear similar on the surface, they are fundamentally different. A *program* is a static sequence of instructions, whereas a *process* is a container for a set of resources used by the threads that execute the instance of the program.

Thread

A *thread* is the entity within a process that Microsoft® Windows® schedules for execution. Without it, the process's program can't run.

Call Stack

A Call Stack is a fundamental data structure, which is used to hold volatile data. The most common use of the call stack is to keep track of function calls and the parameters passed into these functions.

Register

A register is a very fast temporary memory location in the CPU.

General Terms - Continued

- **User Mode**
 - The processor access mode in which applications run
- **Kernel Mode**
 - The processor access mode in which the operating system and privileged programs run
- **Exception**
 - An error condition resulting from the execution of a particular machine instruction.

User Mode

The processor access mode in which applications run. Applications and subsystems run on the computer in user mode. Processes that run in user mode do so within their own virtual address spaces. They are restricted from gaining direct access to many parts of the system, including system hardware, memory that was not allocated for their use, and other portions of the system that might compromise system integrity. Because processes that run in user mode are effectively isolated from the system and other user-mode processes, they cannot interfere with these resources.

Kernel Mode

The processor access mode in which the operating system and privileged programs run. Kernel-mode code has permission to access any part of the system, and is not restricted like user-mode code. It can gain access to any part of any other process running in either user mode or kernel mode.

Exception

An error condition resulting from the execution of a particular machine instruction. Exceptions can be hardware or software-related errors.

General Terms – Continued

- **Interrupt**
 - A condition that disrupts normal thread execution and transfers control to an interrupt handler
- **Interrupt Request Level (IRQL)**
 - The priority ranking of an interrupt
- **Free Build**
 - The retail version of the operating system.
- **Checked Build**
 - The debug version of the operating system.

Interrupt

A condition that disrupts normal command execution and transfers control to an interrupt handler. I/O devices requiring service from the processor usually initiate interrupts.

Interrupt Request Level (IRQL)

The priority ranking of an interrupt. A processor has an IRQL setting that threads can raise or lower. Interrupts that occur at or below the processor's IRQL setting are masked and will not interfere with the current operation. Interrupts that occur above the processor's IRQL setting take precedence over the current operation.

Free Build

The *free build* (or *retail build*) is the end-user version of the operating system. The system and drivers are built with full optimization, debugging asserts are disabled, and debugging information is stripped from the binaries. A free system and driver are smaller and faster, and it uses less memory.

Checked Build

The *checked build* (or *debug build*) serves as a testing and debugging aid in the developing of the operating system and kernel-mode drivers. The checked build contains extra error checking, argument verification, and debugging information that is not available in the free build. A checked system or driver can help isolate and track down driver problems that can cause unpredictable behavior, result in memory leaks, or result in improper device configuration.

General Terms – Continued

- **Paging**
 - A virtual memory operation in which the memory manager transfers pages from memory to disk
- **Paged Pool**
 - A portion of system memory that can be paged to disk
- **Non-Paged Pool**
 - Non-paged pool is a portion of system memory that cannot be paged to disk
- **I/O Request Packet (IRP)**
 - A data structure used to represent an I/O request and control its processing

Paging

A virtual memory operation in which the memory manager transfers pages from memory to disk. *Page fault* occurs when a thread accesses a page that is not in memory.

Paged Pool

A portion of system memory that can be paged to disk. (Note that this term does not only refer to memory that actually has been paged out to the disk — it includes any memory that the operating system is permitted to page.)

Nonpaged Pool

Non-paged pool is a portion of system memory that cannot be paged to disk.

I/O Request Packet (IRP)

A data structure used to represent an I/O request and control its processing. The IRP structure consists of a header and one or more stack locations.

Debugging Terms

- **Debug Session**
 - The act of running a software-debugging program against a computer that has a crashed software component
- **Local Debugging**
 - When the debugger and the application to be debugged reside on the same computer
- **Target Application**
 - The application that will be analyzed using the debugger
- **Attach**
 - Connecting the debugger to a running process

Debug Session

The debug session is the actual act of running a software-debugging program, such as WinDbg, KD, or NTSD, against a computer that has a crashed software component, system service, application, or operating system. The debugging session can also be run against a saved memory dump file for analysis.

Local Debugging

This refers to a debugging session in which the debugger and the application to be debugged reside on the same computer.

Target Application

The application that is failing and will be analyzed using the debugger. Sometimes referred to as the *debuggee*.

Attach

There are some scenarios where the program is already running and there is a need to debug the program. In this case, it is possible to start the debugger and connect to the running program. This is called attaching to a program or process.

Debugging Terms - continued

- **Breakpoint**
 - The point at which the debugger is asked to stop the execution is the breakpoint
- **Disassembly / Assembly Code**
 - Mnemonics representing a series of instructions that the underlying hardware can understand
- **Symbol Files**
 - Files containing the identifiers, such as variables and functions names, created at compile time for use by the debugger
- **Remote Debugging**
 - When the debugger runs on the host computer and the application to be debugged resides on the target computer

Breakpoint

A debugger allows the user to stop the execution of the program at a particular point specified by the user. This is very helpful in viewing the values of the variables at that point. The point at which the debugger is asked to stop the execution is the breakpoint.

Disassembly / Assembly Code

Any microprocessor can understand only the instruction sets defined for that particular processor. When a program is compiled, the compiler will convert the program into a series of instructions that the underlying hardware can understand. The instructions can be represented by mnemonics. This is known as the Assembly code. Using a debugger, it is possible to see the corresponding Assembly code generated for the program. This is the Disassembly code for the program.

Symbol Files

Files containing a map of the source code and all the identifiers, such as variables and functions names, created at compile time for use by the debugger.

Remote Debugging

This refers to a debugging session in which the debugger resides on the host computer and the application to be debugged resides on the target computer.

Debugging Terms – Continued

- **Host Machine**
 - The host machine is the computer that runs the debugging session
- **Target Machine**
 - The computer that needs to be debugged and is the focus of the debugging session
- **Blue Screens, Bug Checks, and Bug Check Codes**
 - When Windows encounters inconsistencies within data necessary for its operation, the operating system shuts down and displays error information on a blue text-mode screen
- **Crash Dump File**
 - System state information written to a file when a bug check occurs

Host Computer

The host computer is the computer that runs the debugging session. In a typical two-system debugging session, the host can be any computer, which can be connected to the target computer. This computer should run a version of Windows that is at least as recent as the one on the target computer.

Target Computer

The target computer is the computer that has experienced the failure of a software component, system service, an application, or of the operating system. This is the computer that needs to be debugged and is the focus of the debugging session. This can be a computer located within a few feet of the computer on which you run the debugging session, or it can be in a completely different location.

Blue Screens, Bug Checks, and Bug Check Codes

When Windows® encounters hardware problems, inconsistencies within data necessary for its operation, or other severe errors, the operating system shuts down and displays error information on a blue-character screen.

This shutdown is known variously as a *bug check*, *kernel error*, *system crash*, *stop error*, or, occasionally, *trap*. The screen display itself is referred to as a *blue screen* or *stop screen*. The most important information on the screen is a message code, which gives information about the crash; this is known as a *bug check code* or *stop code*.

WinDbg or KD can configure the system so that a debugger is automatically contacted when such an error occurs. Alternatively, the system can be instructed to automatically reboot in case of such an error.

Crash Dump File

You can configure the system to write information to a crash dump file on your hard disk whenever a bug check occurs. The file contains information the debugger can use to analyze the error.

Types of Debuggers

- **Character-Based Console Debuggers**
 - **User-Mode Only**
 - NTSD and CDB
 - **Kernel-Mode Only**
 - KD
- **Microsoft Windows-Based Debugger**
 - **Kernel-Mode and User-Mode**
 - WinDbg

CDB and NTSD

CDB and NTSD are console applications, which can debug user-mode programs. These two debuggers are nearly identical, except in the manner in which they are launched.

We will use "CDB" when referring to the capabilities of both CDB and NTSD. *Except as noted, all references to CDB apply equally to NTSD.* There are a few techniques that can only work properly with CDB, or can only work properly with NTSD.

CDB

Microsoft Console Debugger (CDB) is a character-based console program that enables low-level analysis of Windows® user-mode memory and constructs.

CDB is extremely powerful for debugging a program, which is currently running or has recently crashed ("live analysis"), yet simple to set up. It can be used to investigate the behavior of a working application. In the case of a failing application, CDB can be used to obtain a stack trace or to look at the guilty parameters. It works well across a network (using a remote access server), as it is character-based.

CDB can attach to vital subsystem processes, which run during the early boot phase (such as WinLogon or CSRSS), since a graphical application will not work so early in the boot process.

With CDB, you can display and execute program code, set breakpoints, and examine and change values in memory. CDB can analyze binary code by "disassembling" it and displaying assembly instructions. It can also analyze source code directly.

Because CDB can access memory locations through addresses or global symbols, you can refer to data and instructions by name rather than by address, making it easy to locate and debug specific sections of code. You can also display disassembled machine code. CDB supports debugging multiple threads and processes. It is extensible, and can read and write both paged and non-paged memory.

If the target application is itself a console application, the target will share the console window with CDB. To spawn a separate console window for a target console application, use the **-2** command-line option.

Output from CDB can also be redirected to KD.

NTSD

There is a variation of the CDB debugger named Microsoft NT Symbolic Debugger (NTSD). It is identical to CDB in every way, except that it spawns a new text window when it is started, whereas CDB inherits the Command Prompt window from which it was invoked.

Whereas CDB is available as part of the Debugging Tools for Windows package, NTSD is available as part of the Windows system itself. It can be found in the *system32* directory of Windows. *As a result, this documentation may not reflect the current build of the NTSD on your Windows system.*

Like CDB, NTSD is fully capable of debugging both console applications and graphical Windows programs. (The name "Console Debugger" is used to indicate the fact that CDB is classified as a console application; it does not imply that the target application must be a console application.)

The only time when CDB and NTSD are not interchangeable is when you are debugging a user-mode system process. In this case, errors or breaks in this process may cause all console applications to work improperly. Since NTSD is actually capable of running with no console at all, it is required in such cases.

KD

Microsoft Kernel Debugger (KD) is a character-based console program that enables in-depth analysis of kernel-mode activity on Windows.

KD can be used to debug kernel-mode programs and drivers, or to monitor the behavior of the operating system itself. KD also supports multiprocessor debugging.

Typically, the KD tool will not be run on the computer being debugged. Two computers (the host computer and the target computer) are needed for kernel-mode debugging.

KD cannot be used to examine or deposit paged-out memory directly, but the **.pagein** command can be used to page in memory (as long as the target computer is able to run). Most KD commands cannot be targeted to specific processes or threads, as they can in CDB, NTSD, and WinDbg.

Debugging different target platforms

KD is capable of debugging a target computer running on an x86 platform, an Itanium platform, or an AMD x86-64 platform.

The debugger will automatically detect the platform on which the target is running. You do not need to specify the target on the KD command line.

WinDbg

Microsoft Windows Debugger (WinDbg) is a powerful Windows-based debugging tool. It is capable of both user-mode and kernel-mode debugging.

WinDbg provides full source-level debugging for the Windows kernel, kernel-mode drivers, and system services, as well as user-mode applications and drivers.

WinDbg uses the Microsoft® Visual Studio® debug symbol formats for source-level debugging. It can access any public function's names and variables exposed by modules that were compiled with COFF symbol files (such as Windows *.dbg* files).

WinDbg can view source code, set breakpoints, view variables (including C++ objects), stack traces, and memory. It includes a Debugger Command window to issue a wide variety of commands not available through the main window's menu. For kernel-mode debugging, WinDbg typically requires two computers (the host computer and the target computer). It also allows remote debugging of user-mode code.

Module 2: Debugger Installation & Setup

Module 2 Overview

- Debugger Installation
 - Symbol Files
 - Symbol Server
-

Debugger Installation

- **Windows Debugging Resources Web Page**
 - <http://www.microsoft.com/ddk/debugging>
- **Source for:**
 - Windows Debugging Tools package
 - Symbol files
 - Path to Microsoft symbol server
 - Other information and links

Installing Current Debugging Tools

Although various versions of the debugging tools are included in packages such as the Platform SDK, the latest version will always be found at the **Windows Debugging Resources** Web site at <http://www.microsoft.com/ddk/debugging/>.

The Debugging Tools for Windows package is updated frequently. Updating your tools from the Web site assures that you have the most powerful debugging tools possible.

Both 32-bit and 64-bit debugger packages are available. You will only need the 64-bit package if you intend to debug 64-bit user mode programs. The 32-bit package runs on 32-bit systems and can perform both user and kernel mode debugging on 32-bit platforms, and kernel mode debugging with 64-bit target systems.

After the installation is complete, the debugger shortcuts can be found in **Start | Programs | Debugging Tools for Windows**.

Installation Options

The default installation for the Debugging Tools for Windows package includes all the tools, but does not include the *Debugger SDK*. The Debugger SDK is necessary if you intend to write debugger extensions; it includes header files and sample extension code. To include the Debugger SDK in the installation, select the **Custom Install** option during setup, and include the “SDK” item.

Symbol Files

Symbol Files: Overview

- **Public Symbol Files Generally Include**
 - Global variable Names
 - Function names and the address of their entry points
 - FPO data
- **Private Symbol Files Generally Include**
 - Local variable Names
 - Source-line numbers
 - Type information for variables, structures, etc.

When applications, libraries, drivers, or operating systems are linked, the compile and link procedure that creates the .exe, .dll, .sys, and other executable files (collectively known as *binaries*) also creates a number of additional files known as *symbol files*.

Early versions of Windows NT® used symbol files with extension .dbg. Windows 2000 and later versions of Windows NT keep their symbols in files with the extensions .pdb and .dbg. Windows XP and Windows .NET Server use .pdb files exclusively. Symbols for Windows drivers can follow either model, depending on the version of compiler and linker used to build them.

The compiler and the linker control the symbol format. The Visual C++® 5.0 Linker creates both .pdb and .dbg symbol files – the .dbg files it creates are essentially pointers to the .pdb files. The Visual C++ 6.0 Linker, and the Linker supplied with the Driver Development Kit, place all symbol information into .pdb files. Older linkers used only .dbg files.

Symbol files hold a variety of data that is not needed when executing the binaries, but is essential to the debugging process.

Typically, symbol files might contain:

- Names and addresses of global variables
- Function names and the addresses of their entry points
- FPO data
- Names and locations of local variables
- Source file paths and line numbers
- Type information for variables, structures, etc.

The binaries are smaller and faster as a result of keeping these symbol files separate. However, this means that when debugging, you must make sure that the debugger can access the symbol files that are associated with the target you are debugging.

Both interactive debugging and debugging crash dump files require symbols. You must obtain the proper symbols for the code that you wish to debug, and load these symbols into the debugger.

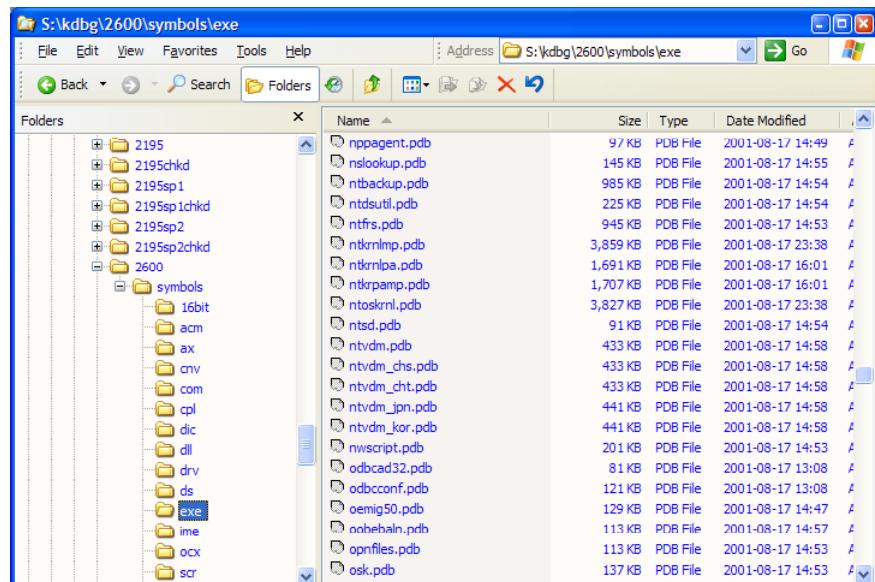
Public vs. Private Symbols

Symbol files can contain either *public* or *private* symbol information. Public symbols include global variables, global symbols such as function entry point names, and FPO data (to be described).

Private symbols include all public symbol information, but also include local variable names, source file paths and line numbers, and type declarations for variables, structures, and so on.

Private symbols are necessary for debugging with the program source files available, while public symbols are usually sufficient for analyzing problems in the absence of source code. The symbols that are freely available from Microsoft for the Windows operating system are always public symbols. Developers who are debugging drivers or applications they have written will usually be working with private symbols for their executables, but will almost always use public symbols for the operating system.

Proper Symbol File Directories



The symbol files on the host machine are required to match the version and build type of Windows installed on the target machine. This fundamental point cannot be overstressed. For example:

- If you are planning to do a kernel mode debug of a Windows 2000 Service Pack 2 target, you need to install the symbols for Windows 2000, and those for Windows 2000 Service Pack 2, on the host system, regardless of which operating system version the host is running.
- If you plan to perform user-mode debugging on the same machine as the target application, then install on that system the symbol files that match the version of Windows running on that system.
- If you are analyzing a memory dump file, then the version of symbol files needed on the debug computer are those that match the version of the operating system that produced the dump file, not necessarily those matching the version of the operating system on the machine running the debug session.

Note If you are going to use your host machine to debug several different target machines, you may need symbols for more than one build of Windows. In this case, be sure to install each symbol collection in a distinct directory path, as illustrated here.

If you are debugging from a Windows machine attached to a network, it may be useful to install symbols for a variety of different builds on a network server. Microsoft® debuggers will accept a network path (`\server\share\dir`) as a valid symbol directory path. This avoids the need for each debugging machine on the network to install the symbol files separately. You may also be able to use a symbol store (to be described).

Symbol files stored on a crashed target machine are not useable by the debugger on the host machine.

Windows Symbol Directory Structure

There are many different types of executable files. The symbol files for all of them use the `.pdb` and, possibly, `.dbg` extensions. For example, the symbol file for `notepad.exe` is `notepad.pdb`. If there were also a `notepad.dll`, its symbol file would also be `notepad.pdb`. To avoid potential name conflicts, it is therefore necessary to keep symbol files for `.exe`, `.dll`, and so on files in separate directories.

The following table lists several of the directories that exist in a standard Windows symbol tree:

Directory	Contains Symbol Files for
ACM	Microsoft Audio Compression Manager files
COM	Executable files (.com)
CPL	Control Panel programs
DLL	Dynamic-link library files (.dll)
DRV	Driver files (.drv)
EXE	Executable files (.exe)
SCR	Screen-saver files
SYS	Driver files (.sys)

Free vs. Checked Symbols

All NT-based operating systems are available in two “builds.”

- The free build, also referred to as the retail build, is the one almost always installed by end users, on server systems, and so on.
- The checked build (also referred to as the debug build) is normally only used by device driver developers as a test platform. Its binaries are larger than those for the free build because they contain additional debugging support and error checking.

Each of these builds has its own symbol files. When debugging, you must use the symbol files that match the build of Windows on the target. As illustrated above, you can use different-named upper level directories to maintain free and checked symbol trees simultaneously on a given system.

Installing Windows Symbol Files

- **To obtain symbol files for Windows**
 - Customer Support Diagnostics CD – RTM version only
 - Service pack download site
 - MSDN subscription CDs for service packs
 - Microsoft Debugging Tools web site – all recent versions
- **Install symbols in version-specific directories**
 - Separate directories for Windows 2000 RTM, Windows 2000 SP1, Windows 2000 SP2, Windows XP RTM, etc.
 - Separate directories for all of the above for free vs. checked build
 - Separate directories for symbols for hotfixes
 - Debugger symbol path specifies latest symbols first – example:
`s:\kdbg\2195sp2;s:\kdbg\2195`

To Obtain Symbol Files for Windows

The “Customer Support Diagnostics” CD-ROM includes symbol files for the base, or “Release To Manufacturing” (RTM) build, of the operating system.

Symbols for Service Pack releases are available from the Windows Update web site, and on the MSDN CD-ROMs that contain the respective Service Packs.

Symbols for all versions, RTM and Service Packs, Free and Checked builds, are available from the *Windows Debugging Resources* web site.

1. Make sure you have at least 500 MB of available space on the disk drive of the host machine.
2. Insert the Customer Support Diagnostics CD, or download and run the appropriate symbol package from the *Windows Debugging Resources* site.
3. For the Customer Support Diagnostics CD, Click on **Install Symbols**, then select either **Install Retail Symbols** (free build) or **Install Debug Symbols** (checked build). The symbols must match the version of the operating system being debugged.
4. Enter the path where the symbols are to be stored, or accept the default path. The default path is `%windir%\symbols`. It is highly recommended that you override the default path, so that you can keep symbols for different operating system versions in different directories.

Installing Symbols in Version-Specific Subtrees

The symbol files for each build (free vs. checked), release (RTM, Service Pack 1, Service Pack 2, etc.) should go into a unique directory subtree. In the example shown previously Windows build numbers (2195 for Windows 2000, 2600 for Windows XP) together with “sp1”, “sp2”, “chkd”, etc. abbreviations were used to form the directory names. The exact names and paths used do not matter.

Installing Symbols in a Single Directory Tree

If you intend to keep your symbols in a single directory tree (*not recommended*), the installation sequence of symbol files should mirror the installation sequence of operating system files:

1. Install the operating system symbol files.
2. Install the symbol files for the currently installed Service Pack (if any).
3. Install the symbol files for any Hot Fixes that were installed after the current Service Pack was installed (if any).

In this way, the symbols present will reflect the files present in the operating system to be debugged.

Installing User-Mode Symbols

If you are going to debug a user-mode application, you also need to obtain and install the symbols for this application as well.

You can debug an application if you have its symbols but not Windows symbols.

However, your results will be much more limited. You will still be able to step through the application code, but any debugger activity (such as getting a stack trace) that requires analysis of the operating system calls made by the application is likely to fail.

Setting the Symbol Path

The debugger must be able to find the symbols you have installed, or else they will be of no use. The debuggers each use a *symbol path* to point to the root of the directory tree containing the symbol files.

This symbol path can contain multiple directory paths, separated by semicolons. If you are debugging an application or driver, which you have written, be sure to include the path to your own symbol files in the debugger's symbol path as well. If you have installed the symbol sets using separate directory subtrees for service packs as recommended, the symbol path should specify the subtree containing the most recent symbols first, and the base level or RTM version of the operating system last.

There are four ways to set the symbol path for WinDbg:

- Set the _NT_SYMBOL_PATH environment variable to point to the root of the directory tree containing the symbols before starting the debugger.
- Use the -y command line option.
- Use the **.sympath** (Set Symbol Path) debugger meta-command.
- Use the “Symbol File Path” command in the File menu, or the Ctrl-S keyboard shortcut, to open the “Symbol Search Path” dialog.

The symbol path must always be set by one of these methods. There is no default path.

Symbol Servers and Symbol Stores

- **Symbol Server**

- An extension of the debugger's symbol path. Lets the debugger access a symbol store for its symbols on an as-needed basis
- Debugging Tools for Windows includes a symbol server called SymSrv (symsrv.dll)

- **Symbol Store**

- A program that indexes symbols so that a symbol server can access them; also, the result of running such a program
- Debugging Tools for Windows includes a symbol store called SymStore (symstore.exe)
- Microsoft has a public symbol store that can be accessed via the internet
- Your site may set up its own symbol store

To set up symbols correctly for debugging can be a challenging task, particularly for kernel debugging. It often requires that you know the names and releases of all products on your computer. The debugger must be able to locate each of the symbol files corresponding to the product releases and service packs.

This can result in an extremely long symbol path consisting of a long list of directories.

To simplify these difficulties in coordinating symbol files, a *symbol server* can be used. A symbol server enables the debuggers to automatically retrieve the correct symbol files without product names, releases, or build numbers. Debugging Tools for Windows contains a symbol server called *SymSrv*.

The symbol server is activated by including a certain text string in the symbol path. Each time the debugger needs to load symbols for a newly loaded module, it calls the symbol server to locate the appropriate symbol files. The symbol server locates the files in a *symbol store*. This is a collection of symbol files, an index, and a tool that can be used by an administrator to add and delete files. The files are indexed according to unique parameters such as the time stamp and image size. Debugging Tools for Windows contains a symbol store tool called *SymStore*.

Using SymSrv

- **Setting the Symbol Path**
 - `srv * \\server\share`
 - `srv * DownstreamStore * \\server\share`
- **Microsoft Public Symbols**
 - `srv * DownstreamStore *`
`http://msdl.microsoft.com/download/symbols`
 - example:
`srv*c:\mysyms*http://msdl.microsoft.com/download/symbols`
- **Compressed Files**
 - Must use a downstream store

SymSrv (*symsrv.dll*) is a symbol server that is included in the Debugging Tools for Windows package.

SymSrv can deliver symbol files from a centralized symbol store. This store can contain any number of symbol files, corresponding to any number of programs or operating systems. The store can also contain binary files (this is useful when debugging minidumps).

The store can contain the actual symbol and binary files, or it can simply contain pointers to symbol files. If the store contains pointers, SymSrv will retrieve the actual files directly from their sources.

SymSrv can also be used to separate a large symbol store into a smaller subset that is appropriate for a specialized debugging task.

Finally, SymSrv can obtain symbol files from an HTTP or HTTPS source using the logon information provided by the operating system. SymSrv supports HTTPS sites protected by smartcards, certificates, and regular logins and passwords.

Setting the Symbol Path

To use this symbol server, *symsrv.dll* must be installed in the same directory as the debugger. The symbol path must be set in one of the following ways:

```
set _NT_SYMBOL_PATH = symsrv*ServerDLL*DownstreamStore* \\Server\Share

set _NT_SYMBOL_PATH = symsrv*ServerDLL* \\Server\Share

set _NT_SYMBOL_PATH = srv*DownstreamStore* \\Server\Share

set _NT_SYMBOL_PATH = srv* \\Server\Share
```

The parts of this syntax are explained in the following table.

Field	Description
symsrv	This keyword must always appear first. It indicates to the debugger that this item is a symbol server, not just a normal

	symbol directory.
<i>ServerDLL</i>	Specifies the name of the symbol server DLL. If you are using the SymSrv symbol server, this will always be symsrv.dll .
srv	This is shorthand for symsrv*symsrv.dll .
<i>DownstreamStore</i>	Specifies a local directory or network share that will be used to cache individual symbol files. If <i>DownstreamStore</i> specifies a directory that does not exist, SymStore will attempt to create it.
<i>\Server\Share</i>	Specifies the server and share of the symbol store.

The *DownstreamStore* parameter can be omitted, as shown in the first syntax example before the table. This parameter is required if you are accessing symbols from an HTTP or HTTPS site, or if you are using compressed files on your store.

If *DownstreamStore* is not included, the debugger will load all symbol files directly from the server.

If *DownstreamStore* is included, the debugger will first look for a symbol file in this location. If the symbol file is not found, the debugger will locate the symbol file from the specified *Server* and *Share*, and then cache a copy of this file in the downstream store. The file will be copied to a subdirectory in the tree under *DownstreamStore* which corresponds to its location in the tree under *\Server\Share*.

The symbol server does not have to be the only entry in the symbol path. If the symbol path consists of multiple entries, the debugger checks each entry for the needed symbol files, in order (from left to right), regardless of whether a symbol server or an actual directory is named.

Here are some examples. To use SymSrv as the symbol server with a symbol store on *\mybuilds\mysymbols*, set the following symbol path:

```
set _NT_SYMBOL_PATH= symsrv*symsrv.dll*\mybuilds\mysymbols
```

To set the symbol path so that the debugger will copy symbol files from a symbol store on *\mybuilds\mysymbols* to your local directory *c:\localsymbols*, use:

```
set _NT_SYMBOL_PATH=symsrv*symsrv.dll*c:\localsymbols*\mybuilds\mysymbols
```

To set the symbol path so that the debugger will copy symbol files from the HTTP site *www.somecompany.com/manysymbols* to a local network directory *\localserver\myshare\mycache*, use:

```
set _NT_SYMBOL_PATH=symsrv*symsrv.dll*\localserver\myshare\mycache*http://www.somecompany.com/manysymbols
```

This last example can also be shortened as such:

```
set _NT_SYMBOL_PATH=srv*\localserver\myshare\mycache*http://www.somecompany.com/manysymbols
```

In addition, the symbol path can contain several directories or symbol servers, separated by semicolons. This allows you to locate symbols from multiple locations (or even multiple symbol servers). If a binary has a mismatched symbol file, the debugger cannot locate it using the symbol server because it checks only for the exact parameters. However, the debugger may find a mismatched symbol file with the correct name, using the traditional symbol path, and successfully load it. Even though the file is technically not the correct symbol file, it might provide useful information.

Public Symbol Store

Microsoft has a Web site that makes Windows symbols publicly available. You can refer directly to this site in your symbol path:

```
set _NT_SYMBOL_PATH=srv*DownstreamStore*http://msdl.microsoft.com/down
load/symbols
```

When using the public symbol store, you should always use a downstream store. Otherwise you will end up downloading the same file several times!

You can add this address to your symbol path automatically by using the **.symfix** (Set Symbol Store Path) command. To append the public symbol store to your existing symbol path, use the following syntax:

```
.symfix+ DownstreamStore
```

Note

To successfully access Microsoft's public symbol store, you will need a fast internet connection. If your internet connection is only 56 Kps or slower, you should install Windows symbols directly onto your hard drive.

For more information on the public symbol store, see <http://www.microsoft.com/ddk/debugging/symbols.asp>.

Compressed Files

SymSrv is compatible with symbol stores that contain compressed files, as long as this compression has been done with the *compress.exe* tool that is distributed with the Platform SDK. Compressed files should have an underscore as the last character in their file extensions (for example, *module1.pd_* or *module2.db_*). For details, see [Using SymStore](#).

If the files on the store are compressed, you must use a downstream store. SymSrv will uncompress all files before caching them on the downstream store.

Deleting the Cache

If you are using a *DownstreamStore* as a cache, you can delete this directory at any time to save disk space.

It is possible to have a vast symbol store that includes symbol files for many different programs or Windows versions. If you upgrade the version of Windows used on your target computer, the cached symbol files will all match the earlier version. These cached files will not be of any further use, and therefore this might be a good time to delete the cache.

How SymSrv Locates Files

SymSrv creates a fully-qualified UNC path to the desired symbol file. This path begins with the path to the symbol store recorded in the *_NT_SYMBOL_PATH* environment variable. The **SymbolServer** routine is then used to identify the name of the desired file; this name is appended to the path as a directory name. Another directory name, consisting of the concatenation of the *id*, *two*, and *three* parameters passed to **SymbolServer**, is then appended. If any of these values is zero, they are omitted.

The resulting directory is searched for the symbol file, or a symbol store pointer file.

If this search is successful, **SymbolServer** passes the path to the caller and returns TRUE. If the file is not found, **SymbolServer** returns FALSE.

Firewalls and Proxy Servers

For information about using SymSrv with firewalls and proxy servers, see the debugger documentation.

Using SymStore

- **Server Administration**
 - SymStore logs all symbol transactions
 - “add” transactions store new symbols
 - “del” transactions delete symbols
- **Compressed Files**
- **The *server.txt* and *history.txt* files**

SymStore (*symstore.exe*) is a tool for creating symbol stores. It is included in the Debugging Tools for Windows package.

SymStore stores symbols in a format that enables the debugger to look up the symbols based on the time stamp and size of the image (for a *.dbg* or executable file), or signature and age (for a *.pdb* file). The advantage of the symbol store over the traditional symbol storage format is that all symbols can be stored or referenced on the same server and retrieved by the debugger without any prior knowledge of which product contains the corresponding symbol.

Note that multiple versions of *.pdb* symbol files (for example, public and private versions) cannot be stored on the same server, because they each contain the same signature and age.

SymStore Transactions

Every call to SymStore is recorded as a transaction. There are two types of transactions: add and delete.

Each time SymStore stores or removes symbol files, a new transaction number is created. If a transaction is deleted, SymStore will read through its transaction file to determine which files and pointers it should delete.

SymStore does not support simultaneous transactions from multiple users. It is recommended that one user be designated "administrator" of the symbol store and be responsible for all **add** and **del** transactions.

For more details, see the debugger documentation.

Module 2 Labs



Module 3: Introduction to Debugger Operation

Module 3 Overview

- Opening a Crash Dump File
 - Debugger Configuration
 - Basic Debugger Commands
 - Interpreting Debugger Displays
-

Opening a Crash Dump File

- **WinDbg Command Line:**
 - `windbg -y SymbolPath -z DumpFileName`
 - `windbg -y SymbolPath -i ImagePath -z MiniDumpFileName`
- **WinDbg Menus:**
 - File | Set Symbol Path (CTRL+S)
 - File | Image File Path (CTRL+I) (for minidumps only)
 - File | Open Crash Dump (CTRL+D)

Kernel-mode memory dump files can be analyzed by WinDbg or KD. These dump files can be debugged on a different Windows version and different processor than the one they were created on.

Starting WinDbg or KD

To analyze a dump file, start WinDbg or KD with the **-z** command-line option:

windbg -y SymbolPath -i ImagePath -z DumpFileName

kd -y SymbolPath -i ImagePath -z DumpFileName

The **-v** option (verbose mode) is also useful.

If WinDbg is already running and is in dormant mode, you can open a crash dump by selecting the **File | Open Crash Dump** menu command or pressing the CTRL+D shortcut key. When the **Open Crash Dump** dialog box appears, enter the full path and name of the crash dump file in the **File name** text box, or use the dialog box to select the proper path and file name. When the proper file has been chosen, press **Open**.

Dump files generally end with the extension *.dmp*. You can use network shares or Universal Naming Convention (UNC) file names for the memory dump file.

If you are analyzing a Kernel Memory Dump or a Small Memory Dump, you may also need to set the executable image path to point to any executable files which may have been loaded in memory at the time of the crash.

The Debugger Command Window

- **The Debugger Command Window Prompt**
 - **KD>**
0: KD>
 - **Editing, Repeating, and Canceling Commands**
 - <Enter> , <CTRL+C> , <CTRL+Break>
 - **Clearing Command Output**
 - **.cls (Clear Screen)**
 - **WinDbg: Many commands also have keyboard shortcut, menu, and toolbar forms**

When running WinDbg, the term *Debugger Command window* refers to the window labeled "Command" in the title bar. This window is split into two panes. The small bottom pane is used to enter text commands. The large upper pane is used to view command output. This window is always open at the beginning of a debugging session. If it is closed, minimized, or hidden, it can be restored by the **View | Command** menu command, pressing the ALT+1 shortcut key, or clicking the following toolbar button:



When running KD, the term *Debugger Command window* refers to the entire text window. Commands are entered at the prompt at the bottom of the screen. If they have any text output, this output will be displayed, and then a prompt will appear.

You can use the UP and DOWN ARROW keys to scroll through the command history. When a previous command appears, it can be edited. Pressing ENTER will cause the old command (or the edited command) to be executed. The cursor does not need to be at the end of the line for this to work correctly.

The Debugger Command Window Prompt

When performing kernel-mode debugging on a target machine that has only one processor, the prompt looks like this:

kd>

However, if the target machine has multiple processors, the number of the current processor will appear before the prompt:

0 : kd>

If no prompt appears in KD, or if the entry pane in WinDbg's Debugger Command window is grayed, no command will be accepted. However, control keys can still be used in KD; menu commands and shortcut keys can still be used in WinDbg.

Editing, Repeating, and Canceling Commands

You can use standard editing keys when you enter a command. Use the UP and DOWN ARROW keys to retrieve previous commands. Edit the current command line with the BACKSPACE, DEL, INS, and LEFT and RIGHT ARROW keys. Press the ESC key to clear the current line.

In NTSD and KD, pressing the ENTER key by itself will repeat the previous command. In WinDbg, this behavior can be enabled or disabled.

If the last command issued is presenting a lengthy display and you wish to cut it off, use the CTRL+C key in NTSD or KD. In WinDbg, use **Debug | Break** or press the CTRL+BREAK shortcut key.

In kernel-mode debugging, commands can also be canceled from the keyboard of the target machine. This is done with CTRL+C for an x86 target.

Clearing Command Output

You can use the **.cls (Clear Screen)** command to clear all the text from the Debugger Command window, clearing the command history entirely.

In WinDbg, this is equivalent to the **Edit | Clear Command Output** menu command.

Setting Paths and Loading Files

- **Symbol Path**
 - _NT_SYMBOL_PATH environment variable
 - -y command line option
 - .sympath (Set Symbol Path) debugger command
 - WinDbg: File | Symbol File Path dialog, or CTRL+S
- **Executable Image Path (minidumps only)**
 - -i command line option
 - .exepath debugger command
 - WinDbg: File | Image File Path dialog, or CTRL+I
- **Source Path**
 - .srcpath
 - WinDbg: File | Source File Path dialog, or CTRL+P

To debug effectively, the debugger should have access to as much information about the target machine or target application as possible.

In many cases, the debugger will be able to locate the files it needs on its own. But in case it cannot do this, you can steer the debugger to the proper files by setting the *executable image path*, the *symbol path*, or the *source path*. You can also load source files at any time, regardless of the source path.

All three of these paths have the same syntax: they consist of multiple directory paths, separated by semicolons. Relative paths are supported, but unless you will always launch the debugger from the same directory, it is preferable to begin each path with a drive letter or a network share. Network shares are also supported.

There are a number of ways of setting and displaying each path, as well as loading and unloading individual files.

The Symbol Path

The *symbol path* specifies the directories in which the binary executable files are located. Some compilers (such as Microsoft® Visual Studio®) place the symbol files in the same directory as the binaries. The symbol files and the checked binaries contain path and filename information. This often allows the debugger to find the symbol files automatically. If you are debugging a user-mode process on the machine where the executable was built, and if the symbol files are still in their original location, the debugger will be able to locate the symbol files without this path being set.

In most other cases, you will need to set the symbol path to point to your symbol file locations.

For each directory in the symbol path, the debugger will look in three directories. For instance, if the symbol path includes the directory *c:\MyDir*, and the debugger is looking for symbol information for a DLL, the debugger will first look in *c:\MyDir\symbols\dll*, then in *c:\MyDir\dll*, and finally in *c:\MyDir*. It will repeat this for each directory in the symbol path. Finally, it will look in the current directory, and then the current directory with *\dll* appended to it. (The debugger will append *dll*, *exe*, or *sys*, depending on what binaries it is debugging.)

However, since symbol files all have date and time stamps, there is no danger that the debugger will use the wrong symbols just because they are found first in this sequence. The debugger will always look for the symbols which match the time stamp on the binaries it is debugging.

The debugger's default behavior is to use *lazy symbol loading*. This means that symbols are not actually loaded until they are needed. However, if the symbol path is changed, all *already-loaded* symbols are immediately reloaded.

Lazy symbol loading can be turned off in NTSD and KD with the **-s** command-line option. Symbol loading can also be forced with the **.reload /f (Reload Symbols)** command.

There are several methods of controlling the symbol path:

- (*Before starting the debugger*) The `_NT_SYMBOL_PATH` and `_NT_ALT_SYMBOL_PATH` environment variables can be used to set the path. The symbol path is created by appending the `_NT_SYMBOL_PATH` *after* the `_NT_ALT_SYMBOL_PATH`. (Normally, the entire path is set through the `_NT_SYMBOL_PATH`. However, you might want to use `_NT_ALT_SYMBOL_PATH` to override these settings in special cases—for instance, if you have private versions of shared symbol files.)
- (*When starting the debugger*) The **-y** command-line option can be used to set the path.
- The **.sympath** command can be used to display, set, change, or append to the path.
- (*WinDbg only*) The **File | Symbol File Path** menu command or the CTRL+S shortcut key can be used to display, set, change, or append to the path.

The Executable Image Path

The *executable image path* specifies the directories in which the binary executable files are located.

In most cases, the debugger will know the location of the executables, so this path does not need to be set.

However, there are cases where this path is needed. For instance, if you are analyzing a kernel-mode dump file, the Kernel Memory Dump and Small Memory Dump will not contain all the executables residing in memory at the time of the crash. Also, anything paged to disk at the time of the crash will not be included in the dump file. Setting this path allows the debugger to find these binary files.

The debugger searches the executable image path recursively: each directory listed in this path will have its entire subdirectory tree searched.

There are several methods of controlling the executable image path:

- (*Before starting the debugger*) The `_NT_EXECUTABLE_IMAGE_PATH` environment variable can be used to set the path.
- (*When starting the debugger*) The **-i** command-line option can be used to set the path.
- The **.exepath** command can be used to display, set, change, or append to the path.
- (*WinDbg only*) The **File | Image File Path** menu command or the CTRL+I shortcut key can be used to display, set, change, or append to the path.

The Source Path

The *source path* specifies the directories in which the C and C++ source files are located.

If you are debugging a user-mode process on the machine where the executable was built, and if the source files are still in their original location, the debugger will be able to locate the source files automatically. In most other cases, you will need to set the source path or load the individual source files.

When you are performing Remote Debugging Through the Debugger, the source path is used by the Debugging Server. If you are using WinDbg as your debugger, each Debugging Client has its own *local source path* as well. All source-related commands will access the source files on the local machine. So the proper paths have to be set on any client or server which wishes to use source commands.

This multiple-path system also allows a Debugging Client to use source-related commands without actually sharing the source files with other clients or with the server. This is useful if there are private or confidential source files which one of the users has access to.

There are several methods of controlling the source path and local source path:

- (*Before starting the debugger*) The _NT_SOURCE_PATH environment variable can be used to set the source path.
- (*When starting the debugger*) The **-srcpath** command-line option can be used to set the source path.
- The **.srcpath** command can be used to display, set, change, or append to the source path.
- (*WinDbg only*) The **.lsrcpath** command can be used to display, set, change, or append to the local source path.
- (*WinDbg only*) The **File | Source File Path** menu command or the CTRL+P shortcut key can be used to display, set, change, or append to either the source path or the local source path.

There are also several methods of directly opening or closing a source file:

- The **LSF (Load or Unload Source File)** command can be used to open or close a source file.
- (*WinDbg only*) The **.open (Open Source File)** command can be used to open a source file.
- (*WinDbg only*) The **File | Open Source File** menu command or the CTRL+O shortcut key can be used to open a source file. The following toolbar button can also be used:



- (*WinDbg only*) The **File | Recent Files** menu command can be used to open one of the four most recently-opened source files.
- (*WinDbg only*) To close a source file, select **File | Close** or simply click on the "X" in the corner of the Source window.

Checking for Symbol Problems

- **Symbol errors may appear when you begin**
- **!sym noisy — gives verbose symbol information**
- **.reload — reloads symbols**
 - **.reload Module** — reloads symbols for one module
 - **.reload** — reloads all symbols
- You must always use **.reload** after you change the symbol path or fix a symbol error — the debugger doesn't automatically reload your symbols!

The debugger must have access to symbol information that matches the executables of the system being debugged. This point cannot be overstressed.

When the debugger first opens a crash dump file (or at the beginning of a live kernel debugging session, as described later), look carefully in the command window for any indication of symbol file problems. This may be in the form of an obvious error message from the debugger, such as

```
*** ERROR: Symbol file could not be found. Defaulted to export symbols
for ntoskrnl.exe -
```

or

```
***** Kernel symbols are WRONG. Please fix symbols to do analysis.
```

In such cases you will need to resolve the symbol problems before proceeding. The following debugger commands can assist with this:

!sym noisy

Puts the debugger into “noisy” symbol loading mode. All file names attempted to be opened in the search for symbols will be reported in the command window.

.reload [*ModuleName*]

Directs the debugger to attempt to load the symbols again, for the indicated module if specified, or for the entire target system. For example, **.reload nt** refers to the *ntoskrnl.exe* module, which is the most important for kernel debugging.

For example, here we deliberately set the symbol path to that for Windows NT 4.0 (build 1381) and then tried to open a dump file created on a Windows 2000 system:

```
kd> .sympath S:\kdbg\1381
Symbol search path is: S:\kdbg\1381
kd> .reload nt
DBGHELP: ntoskrnl.exe is stripped. Searching for dbg file
DBGHELP: S:\kdbg\1381\ntoskrnl.dbg - file not found
DBGHELP: S:\kdbg\1381\symbols\exe\ntoskrnl.dbg - path not found
DBGHELP: S:\kdbg\1381\exe\ntoskrnl.dbg - path not found
DBGHELP: ntoskrnl.exe missing debug info. Searching for pdb anyway
DBGHELP: S:\kdbg\1381\ntoskrnl.pdb - file not found
DBGHELP: S:\kdbg\1381\symbols\exe\ntoskrnl.pdb - file not found
DBGHELP: S:\kdbg\1381\exe\ntoskrnl.pdb - file not found
```

```

DBGHELP: ntoskrnl.pdb - file not found
*** ERROR: Symbol file could not be found. Defaulted to export symbols
for ntoskrnl.exe -
DBGHELP: nt - export symbols

```

Since Windows NT 4.0 does not use .pdb files, the debugger indicates that it can't find the file it's looking for, under any of the possible path variations. "Defaulted to export symbols" tells us that the debugger is, by default, using the symbols exported by the ntoskrnl.exe file on the local system. This will usually not match the target being debugged.

We then mistakenly set the symbol path to our tree of Windows 2000 symbols, and try again:

```

kd> .sympath s:\kdbg\2600
Symbol search path is: s:\kdbg\2600
kd> .reload nt
DBGHELP: ntoskrnl.exe is stripped. Searching for dbg file
DBGHELP: s:\kdbg\2600\ntoskrnl.dbg - file not found
DBGHELP: s:\kdbg\2600\symbols\exe\ntoskrnl.dbg - file not found
DBGHELP: s:\kdbg\2600\exe\ntoskrnl.dbg - path not found
DBGHELP: ntoskrnl.exe missing debug info. Searching for pdb anyway
DBGHELP: s:\kdbg\2600\ntoskrnl.pdb - file not found
DBGHELP: s:\kdbg\2600\symbols\exe\ntoskrnl.pdb - unknown pdb sig OK
DBGENG: ntoskrnl.exe has mismatched symbols - type ".hh dbgerr003" for
details
DBGHELP: nt - public symbols - s:\kdbg\2600\symbols\exe\ntoskrnl.pdb

```

It fails again, but for a different reason. This time it found an ntoskrnl.pdb, but doesn't recognize the "signature," because it's a different format used for a different operating system version.

In the next try, we have the operating system version correct, but we mistakenly point the debugger at the symbols for the checked build, when the target system was running the free build:

```

kd> .sympath s:\kdbg\2195chkd
Symbol search path is: s:\kdbg\2195chkd
kd> .reload nt
DBGHELP: ntoskrnl.exe is stripped. Searching for dbg file
DBGHELP: s:\kdbg\2195chkd\ntoskrnl.dbg - file not found
DBGHELP: s:\kdbg\2195chkd\symbols\exe\ntoskrnl.dbg - OK
DBGHELP: s:\kdbg\2195chkd\ntoskrnl.pdb - file not found
DBGHELP: s:\kdbg\2195chkd\symbols\exe\ntoskrnl.pdb - OK
*** WARNING: symbols timestamp is wrong 0x384d9b17 0x384d4cf0 for
ntoskrnl.exe
DBGHELP: nt - public symbols - s:\kdbg\2195chkd\symbols\exe\ntoskrnl.pdb

```

This is a common sort of error message for mismatched symbols: The "timestamp" is wrong, indicating that although the symbol file is of the correct format and so on, it isn't the correct version. This can frequently occur due to service packs or hotfixes that were installed on the target system, but for which symbol files do not exist in the debugger's symbol path.

Finally we set the symbol path to the correct one for the dump:

```

kd> .sympath s:\kdbg\2195
Symbol search path is: s:\kdbg\2195
kd> .reload nt
DBGHELP: ntoskrnl.exe is stripped. Searching for dbg file
DBGHELP: s:\kdbg\2195\ntoskrnl.dbg - file not found
DBGHELP: s:\kdbg\2195\symbols\exe\ntoskrnl.dbg - OK
DBGHELP: s:\kdbg\2195\ntoskrnl.pdb - file not found
DBGHELP: s:\kdbg\2195\symbols\exe\ntoskrnl.pdb - OK
DBGHELP: nt - public symbols - s:\kdbg\2195\symbols\exe\ntoskrnl.pdb

```

The "OK" tells us that we did the right thing.

We would likely then use a **.reload** command with no parameters to force the debugger to reload symbols for all modules in the target system from the new symbol file path.

Messages of the form

```
*** ERROR: Symbol file could not be found. Defaulted to export symbols  
for XYZDRIVER.SYS -
```

Probably do *not* indicate symbol file problems, but rather the absence of symbol files for the indicated third-party driver.

Sometimes a **.reload** is not sufficient – without proper symbols the debugger sometimes has difficulty performing the reload of symbol information! In such cases you will need to stop the debugging session completely, set the symbol path properly, then open the dump file again.

Ending the Debugging Session

- **Ending the Session**
 - **Debug | Stop Debugging**
- **Exiting WinDbg**
 - **Q (Quit)**
 - **File | Exit**

You can exit any debugger at any time during the debugging session.

Ending the Session but not Exiting

When performing user-mode debugging in WinDbg, you can end the debugging session without exiting WinDbg. This is done by selecting the **Debug | Stop Debugging** menu command. This can also be done by pressing the SHIFT+F5 shortcut key or by using the following button from the toolbar:



You will be prompted to save the workspace for the current session, and then WinDbg will return to dormant mode. At this point, all starting options are available to you: you can begin debugging a running process, spawn a new process, attach to a target machine, open a crash dump, or connect to a remote debugging session.

Exiting WinDbg

When you are using WinDbg to do kernel debugging, there are two ways to exit.

Issuing a **Q (Quit)** command in WinDbg will save the log file, end the debugging session, and exit the debugger. The target machine will remain locked.

Selecting the **File | Exit** menu command, or using the ALT+F4 shortcut key, will allow you to exit the debugger even if there is no command prompt. This allows you to let the target machine keep running after the debugger is terminated.

Displaying Memory

- **Reading memory by virtual address**
 - D Display memory range
 - DD *range* display as DWORDs (32 bits)
 - DW *range* display as words (16 bits)
 - DB *range* display as bytes
- ***range* can be:**
 - address displays 32 items
 - address1 address2 display range
 - address1 L *n* display *n* items
- **address can be symbol, constant, expression**
- **WinDbg only: Memory window (ALT+5)**

There are several commands which allow you to access specified memory addresses or address ranges.

The following commands can read or write memory in a variety of formats. These include hexadecimal bytes, words, double words, and quad-words; short, long, and quad integers and unsigned integers; 10, 16, 32, and 64 byte real numbers, and ASCII characters.

- The **D*** (**Display Memory**) command can display the contents of a specified memory address or range.
- (*WinDbg only*) The Memory window can display the contents of a specified memory range.

The following commands can be used to deal with more specialized data types:

- The **DT** (**Dump Type**) command can look up a great variety of data types, and display data structures which have been created by the program being debugged. This is a highly versatile command, and has many variations and options.
- The **Ds** (**Display String**) command can display a STRING or ANSI_STRING data structure.
- The **DS** (**Display String**) command can display a UNICODE_STRING data structure.
- The **DL** (**Dump Linked List**) command can trace and display a linked list.
- The **D*S** (**Display Words and Symbols**) command can look up double-words or quad-words which may contain data that is close to the values of known symbol information, and then display the data and the associated symbol information.

The following commands can be used to manipulate entire memory ranges:

- The **C** (**Compare Memory**) command can compare the contents of two memory ranges.
- The **S** (**Search Memory**) command can search for a specified byte pattern in memory.

In most cases, these commands will interpret their parameters in the current radix. Therefore, hexadecimal addresses should be prefixed by **0x** if this radix is not 16. However, the display output of these commands is usually in hexadecimal, regardless of the current radix. (See the individual command pages for details.) The Memory window displays integers and real numbers in decimal, and displays other formats in hexadecimal.

To change the default radix, use the **N (Set Number Base)** command. To quickly convert numbers from one base to another, use the **? (Evaluate Expression)** or the **.formats (Show Number Formats)** command.

Observing Memory via the Memory Window

WinDbg has an additional way of displaying memory: The Memory window. Use the View | Memory command, or the keyboard shortcut Alt-5, or the click the following toolbar button:



The Memory window can maintain a continuous display of any region of memory you choose. As with the **D*** commands the starting address of the region can be given as a constant or a symbol. The Memory window can display the region as many different types of data, such as hexadecimal longwords, signed short integers, 64-bit floating point numbers, and so on.

Displaying Registers and Flags

- **Registers: Fast storage locations within the CPU**
- **Identified by name, not by address**
 - EAX, EBX, EBP, ESP, EIP, etc.
- **R (Registers) command**
- **(WinDbg only) Registers window**
 - View | Registers
 - or ALT+4
- **Flags register (EFL): Bits set as results of other operations**

Registers are small volatile storage units, generally 32 or 64 bits wide, located on the CPU. Unlike ordinary memory, they are addressed by unique names rather than addresses. Many of the registers are dedicated to specific uses, whereas others are available for general use by programs.

A common cause of bugs is that a register contains the wrong data. As we will see in later material, it is often possible to isolate such bugs to the failing module through careful interpretation of the debugger's register displays.

The x86, Itanium, and AMD x86-64 processors each have a different collection of registers available.

There are two ways to display the CPU registers:

- The **R (Registers)** command can be used to display or control the registers. This display can be customized by using a number of options, or by using the **Rm (Register Mask)** command.
- *(WinDbg only)* The Registers window can be used to display or control the registers. It can be customized to display the registers in any order.

On an x86 processor, the **R** command also displays a number of one-bit registers known as *flags*. Flags are individual bits, most of which are automatically set or cleared as side effects of operations. For example, if the result of any arithmetic or logic operation is zero, the flag called ZF (zero flag) will be set; any such operations that give non-zero results will result in this flag being cleared. The more important flags are displayed in the normal register output in the command window, and as individual bits (OF, DF, IF, TF, SF, ZF, AF, PF, CF) in the Registers window. The EFL register is a 32-bit register containing all of the flags treated as a single longword.

The Call Stack

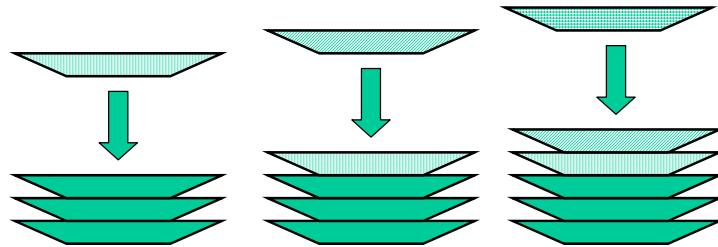
Stacks

- **Uses:**
 - Track function calls and parameters
 - Storing register contents for later restoration
 - Local variables
- **Exotic uses:**
 - Reversing the order of a string of characters
 - Parsing expressions

The stack is one of the fundamental data structures used in computer programming. The stack structure can be used to perform useful tasks such as reversing the order of a string of characters, temporarily holding data to be retrieved later, and for vastly more complex tasks such as parsing expressions. The most common use of a stack on a computer is to keep track of function calls and parameters passed to these functions.

Adding Items

In a cafeteria, plates are added to the top of the stack.

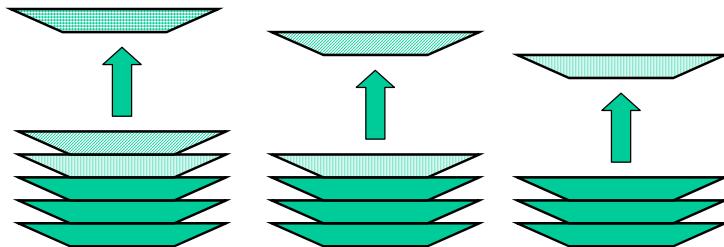


A computer stack operates similarly to a stack of plates in a cafeteria. Items are always added and removed from the current top of the stack. In this plate example, multiple plates could be picked up at one time in order to access plates lower on the stack. Once plates were removed however the new top of the stack would now be this location.

Removing Items

In order to retrieve plates on the bottom of the stack, all the previously added plates are removed first.

LIFO Data Structure



As plates are needed, they are picked up from the top of the stack of plates. Computer stacks also retrieve items from the current top of the stack. This behavior of always adding or removing items from the top of the stack is referred to as Last-In/First-Out or LIFO. A stack is a LIFO data structure.

Thread Stacks

- Two stacks for every thread
 - One for user mode (1 MB, pageable)
 - One for kernel mode (12 KB, normally nonpageable)
- ESP register (stack pointer) always contains address of last item pushed onto stack
- "r" Command – shows ESP register (among others)

```
kd> r
EAX=01b40000 EBX=7ffd000 ECX=00000000
EDX=00000001 ESI=00142851 EDI=00000000

EIP=01b4211c ESP=0012ff18 EBP=0012ffc0
EFL=00000206 CS=001b DS=0023 ES=0023 SS=0023
FS=0038 GS=0000
```

Having discussed basic stack operations, the focus is now on the specifics of stack creation and operation in Win32 on x86 platforms. When a thread is created, memory is allocated for two stacks: A one megabyte, pageable area for use while the thread is in user mode, and a 12 Kbyte area of nonpageable memory for the kernel mode stack.

The ESP register always contains the address of the last data that was pushed onto the stack. The remainder of the currently used area of the stack can be found at successively higher addresses from that shown in the ESP register.

To examine the current value of the stack pointer, use the "r" command in the debugger. The ESP register holds the current value of the stack pointer.

```
0: kd> r
eax=ffdfff13c ebx=0000007f ecx=80539700 edx=00000000
esi=00000000 edi=00000000
eip=804f087d esp=805366c8 ebp=805366e0 iopl=0          nv up
di ng nz na po nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000
efl=00000086
nt!KeBugCheckEx+19:
804f087d 5d          pop     ebp
```

To display the current stack area we could use the ESP register's contents with the **D*** command:

```
0: kd> dd 805366c8
805366c8 0000007f 00000008 80042000 00000000
805366d8 00000000 00000000 00000000 8052d188
805366e8 0000007f 00000008 80042000 00000000
805366f8 00000000 00000000 00000000 00000000
80536708 00000000 00000000 00000000 00000000
80536718 00000000 00000000 00000000 00000000
80536728 00000000 00000000 00000000 00000000
80536738 00000000 00000000 00000000 00000000
```

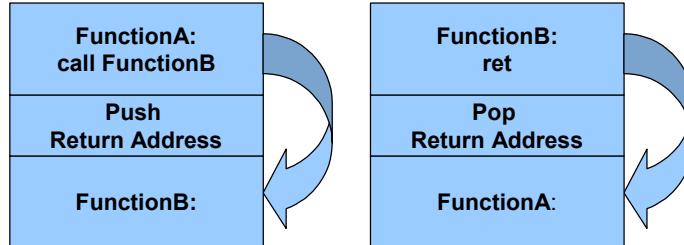
In fact the **D*** command can use the ESP register's name directly, as shown here:

```
0 : kd> dd esp
805366c8 0000007f 00000008 80042000 00000000
805366d8 00000000 00000000 00000000 8052d188
805366e8 0000007f 00000008 80042000 00000000
805366f8 00000000 00000000 00000000 00000000
80536708 00000000 00000000 00000000 00000000
80536718 00000000 00000000 00000000 00000000
80536728 00000000 00000000 00000000 00000000
80536738 00000000 00000000 00000000 00000000
```

Displaying the stack is usually much easier than this, however. Analyzing stack contents is an important aspect of debugging, so the debugger includes some specialized commands to make this easier., as we will illustrate shortly.

Nested Function Calls

- When a function calls another function, the call stack is used.
- A return address is pushed when a CALL instruction is executed.
- The return address is popped when a RET instruction is executed.



Rarely is a program written as one long function. Typically, other functions are written to handle specific program functionality and are called as needed from the main program. Here is a small C program that uses functions found in the standard C runtime library.

When main() calls Addemup() in the example, the address of the instruction immediately following the CALL instruction in main() is pushed onto the stack, and the instruction pointer is set to the address of Addemup(). Therefore, execution can continue inside the new function.

C Code:

```
#include <stdio.h>

int Addemup(int, int);

void main(void)
{
    int x = 5;
    int y = 10;
    int z = 0;
    z = Addemup(x, y);
    printf("z= %i\n", z);
}

int Addemup(int a, int b)
{
    int c = 0;
    c=a+b;
    return(c);
}
```

This program initializes three local variables, calls a function called Addemup() with two of these local variables as arguments. Addemup() adds these two values together and returns this value to main(), and main() calls printf() with the result. However, this means that the program's initial function, main() must call a function named Addemup() and

somewhat return back to main() to finish the rest of the main() function. How is this accomplished? The answer lies in the use of the stack as a temporary placeholder.

Assembly Example:

Here's the Assembly for the C program:

```
addemup!main:
00401000 55          push    ebp
00401001 8bec         mov     ebp,esp
00401003 83ec0c       sub     esp,0xc
00401006 c745fc050000000 mov    dword ptr [ebp-0x4],0x5
0040100d c745f80a0000000 mov    dword ptr [ebp-0x8],0xa
00401014 c745f4000000000 mov    dword ptr [ebp-0xc],0x0
0040101b 8b45f8       mov    eax,[ebp-0x8]
0040101e 50          push    eax
0040101f 8b4dfc       mov    ecx,[ebp-0x4]
00401022 51          push    ecx
00401023 e81b0000000 call   addemup!Addemup (00401043)
00401028 83c408       add    esp,0x8
0040102b 8945f4       mov    [ebp-0xc],eax
0040102e 8b55f4       mov    edx,[ebp-0xc]
00401031 52          push    edx
00401032 6830704000  push   0x407030
00401037 e8220000000 call   addemup!printf (0040105e)
0040103c 83c408       add    esp,0x8
0040103f 8be5         mov    esp,ebp
00401041 5d          pop    ebp
00401042 c3          ret
addemup!Addemup:
00401043 55          push    ebp
00401044 8bec         mov     ebp,esp
00401046 51          push    ecx
00401047 c745fc000000000 mov    dword ptr [ebp-0x4],0x0
0040104e 8b4508       mov    eax,[ebp+0x8]
00401051 03450c       add    eax,[ebp+0xc]
00401054 8945fc       mov    [ebp-0x4],eax
00401057 8b45fc       mov    eax,[ebp-0x4]
0040105a 8be5         mov    esp,ebp
0040105c 5d          pop    ebp
0040105d c3          ret
```

Assembly Walk-through:

This is a look at the stack prior to setting up for the call to Addemup():

```
0:000> dd esp
0012ff74 00000000 0000000a 00000005 0012ffc0
0012ff84 00401146 00000001 00300ec0 00300e30
0012ff94 77fc9816 00270718 7ffd0000 80100000
0012ffa4 be554d00 0012ff94 00000000 0012ffe0

0012ff74 00000000 <= mov dword ptr [ebp-0xc],0x0 <= esp
0012ff78 0000000a <= mov dword ptr [ebp-0x8],0xa
0012ff7c 00000005 <= mov dword ptr [ebp-0x4],0x5
0012ff80 0012ffc0 <= push ebp
```

Here we push the two parameters onto the stack right to left and then make the call to Addemup():

```
0040101b 8b45f8       mov    eax,[ebp-0x8]
0040101e 50          push    eax
0040101f 8b4dfc       mov    ecx,[ebp-0x4]
00401022 51          push    ecx
00401023 e81b0000000 call   addemup!Addemup (00401043)
```

Here is the stack following the previous five lines of code:

```
0:000> dd esp
0012ff68 00401028 00000005 0000000a 00000000
0012ff78 0000000a 00000005 0012ffc0 00401143
0012ff88 00000001 00300ec0 00300e30 00000000
0012ff98 00000000 7ffd0000 80100000 be917d00
```

```
0012ff68  00401028 <= return address
0012ff6c  00000005 <= push ecx (argument 1)
0012ff70  0000000a <= push eax (argument 2)
0012ff74  00000000 <= mov dword ptr [ebp-0xc],0x0 <= esp
0012ff78  0000000a <= mov dword ptr [ebp-0x8],0xa
0012ff7c  00000005 <= mov dword ptr [ebp-0x4],0x5
0012ff80  0012ffc0 <= push ebp

0:000> r
eax=0000000a ebx=7ffd000 ecx=00000005 edx=00000003 esi=00000000
edi=00000000
rip=00401043 esp=0012ff68 ebp=0012ff80 iopl=0      nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000          efl=00000216
```

Notice when the call was made to addemup!Addemup() the return address was pushed onto the stack and esp decremented. The instruction pointer is updated as well to point to the first line of code in Addemup().

Viewing the Call Stack

- **K (Display Stack Backtrace)**
 - Base Pointer, Return Address, Name of Function
- **KB (Display Stack Backtrace)**
 - Base Pointer, Return Address, Name of Function, First Three Parameters
- **KV (Display Stack Backtrace)**
 - Base Pointer, Return Address, Name of Function, First Three Parameters, FPO Infrmation
- **KD (Display Stack Backtrace)**
 - Raw Stack Data, No Formatting
- **WinDbg: Call Stack Window (ALT+6)**

The "K" commands simplify the re-creation of the call stack by automatically reading information from the stack and displaying it as formatted text. Several forms of the command exist, each one dumping the basic plus additional information. As functions are executed and call other functions, a call stack is created in stack memory. The "K" command walks stack memory and tries to determine which functions are being executed and displays the function calls in the order that they have been executed.

"K" Command Example:

Typing 'k' dumps basic stack information including the symbolic name of the functions in the current call stack.

```
ChildEBP RetAddr
0012ff64 00401028 addemup!Addemup+0x11
0012ff80 00401143 addemup!main+0x28
0012ffc0 77e87903 addemup!mainCRTStartup+0xb4
0012fff0 00000000 KERNEL32!BaseProcessStart+0x3d
```

"KB" Command Example:

The "KB" command displays the first three DWORDS of information passed as parameters to each function on the stack.

```
ChildEBP RetAddr  Args to Child
0012ff64 00401028 00000005 0000000a 00000000 addemup!Addemup+0x11
0012ff80 00401143 00000001 00300ec0 00300e30 addemup!main+0x28
0012ffc0 77e87903 77fc9816 00270718 7fdf000 addemup!mainCRTStartup+0xb4
0012fff0 00000000 0040108f 00000000 000000c8
KERNEL32!BaseProcessStart+0x3d
```

"KV" Command Example:

The "KV" command is the same as "KB", but dumps additional information regarding FPO data.

```
ChildEBP RetAddr  Args to Child
0012ff64 00401028 00000005 0000000a 00000000 addemup!Addemup+0x11
0012ff80 00401143 00000001 00300ec0 00300e30 addemup!main+0x28
0012ffc0 77e87903 77fc9816 00270718 7fdf000 addemup!mainCRTStartup+0xb4
0012fff0 00000000 0040108f 00000000 000000c8
KERNEL32!BaseProcessStart+0x3d (FPO: [Non-Fpo])
```

Other "K" options

The "KP" command displays the full parameter list for each function called in the stack trace. However, this requires full symbol information to work properly.

You can also add "N" to display the frame numbers (for example, "KVN").

The Calls Window

In WinDbg, the **Calls window** displays call stack information. It can be customized to display different stack data. It can also be used to quickly jump to the corresponding function in a Source or Disassembly window.

Changing the Context

You can use the **.trap**, **.thread**, **.cxr**, **.exr**, and **.ecxr** commands to focus the debugger on a different thread context. This allows you to display the stack of different threads with the "K" commands or the Calls window. This will be discussed in more detail in Module 9.

Module 3 Labs



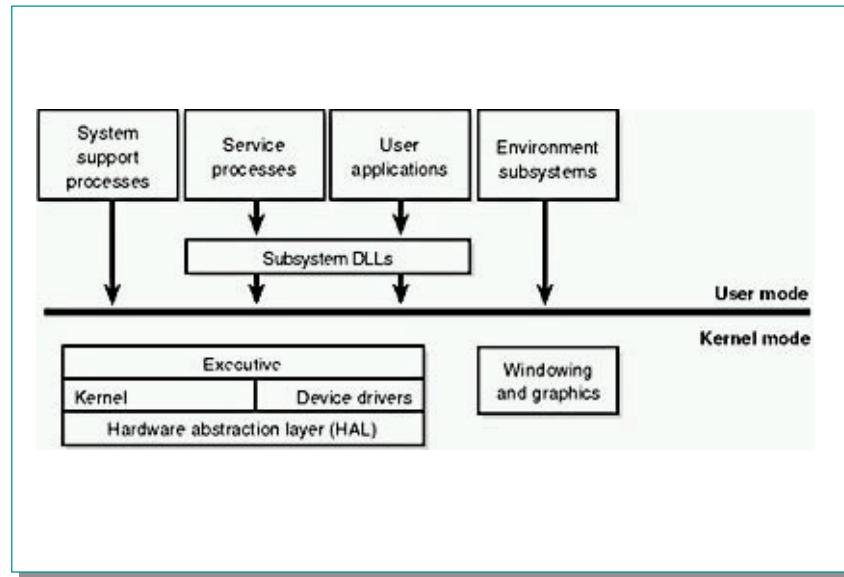
Module 4: Key Concepts & Data Structures

Module 4 Overview

- Component Overview
- Processes & Threads
- Trap Dispatching
- Memory Management
- System I/O
- Related Debugger Commands

Key Concepts and Data Structures

Component Overview



In the above slide, notice the line dividing the user-mode and kernel-mode parts of the Windows operating system. The boxes above the line represent user-mode processes, and the components below the line are kernel-mode operating system services. Threads executing in user mode have access only to a protected, per-process address space. When executing in kernel mode they have access to system space. Thus, system support processes, service processes, user applications, and environment subsystems each have their own private process address space.

Notice the "Subsystem DLLs" box below the "Service processes" and "User applications" boxes. Under Windows, user applications don't call the native operating system services directly; rather, they go through one or more *subsystem dynamic-link libraries* (DLLs). The role of the subsystem DLLs is to translate a documented function into the appropriate internal (and undocumented) Windows system service calls. This translation might or might not involve sending a message to the environment subsystem process that is serving the user application.

Processes

- **System support processes** that are not true Windows services (since they are not started by the service control manager) – for example, the logon process and the session manager.
- **Service processes** that host Win32 services, such as the Task Scheduler and Spooler services. Many Windows server applications have components that run as services.
- **User applications**, which can be one of five types: Win32, Windows 3.1, MS-DOS, POSIX, or OS/2 1.2.
- **Environment subsystems**, which expose the native operating system services to user applications through a set of callable functions. Windows ships with three environment subsystems: Win32, POSIX, and OS/2.

The four basic types of user-mode processes are described as follows:

- Fixed (or hardwired) *system support processes*, such as the logon process and the session manager, that are not Win32 services (that is, not started by the service control manager).
- *Service processes* that host Win32 services, such as the Task Scheduler and Spooler services. Many server applications, such as Microsoft SQL Server and Microsoft Exchange Server, also include components that run as services.
- *User applications*, which can be one of three types: Win32, Windows 3.1, or MS-DOS.
- *Environment subsystems*, which expose the native operating system services to user applications through a set of callable functions, thus providing an operating system *environment*, or personality. Windows NT and Windows 2000 ship with three environment subsystems: Win32, POSIX, and OS/2. Windows XP and later operating systems include only Win32.

Kernel-mode Components

- The ***executive*** contains the base operating system services, such as memory management, process and thread management, security, I/O, and interprocess communications
- The ***kernel*** consists of low-level operating system functions, such as thread scheduling, interrupt and exception dispatching, and multiprocessor synchronization
- ***Device drivers*** include both hardware device drivers that translate user I/O function calls into specific hardware device I/O requests as well as file system and network drivers
- The ***hardware abstraction layer (HAL)*** is a layer of code that isolates the kernel, device drivers, and the rest of the Windows executive from platform-specific hardware
- The ***windowing and graphics system*** implements the graphical user interface (GUI) functions, such as dealing with windows, user interface controls, and drawing

The kernel-mode components of Windows include the following:

- The ***executive*** contains the base operating system services, such as memory management, process and thread management, security, I/O, and interprocess communication.
- The ***kernel*** consists of low-level operating system functions, such as thread scheduling, interrupt and exception dispatching, and multiprocessor synchronization. It also provides a set of routines and basic objects that the rest of the executive uses to implement higher-level constructs.
- ***Device drivers*** include both hardware device drivers that translate user I/O function calls into specific hardware device I/O requests as well as file system and network drivers.
- The ***hardware abstraction layer (HAL)*** is a layer of code that isolates the kernel, device drivers, and the rest of the executive from platform-specific hardware differences (such as differences between motherboards).
- The ***windowing and graphics system*** implements the graphical user interface (GUI) functions (better known as the Win32 USER and GDI functions), such as dealing with windows, user interface controls, and drawing.

The following table lists the filenames of the core components. (You'll need to know these filenames because we'll be referring to some system files by name.)

Name of the file on the system disk	Components
Ntoskrnl.exe	Executive and kernel
Ntkrnlpa.exe	Executive and kernel with support for Physical Address Extension (PAE), which allows addressing of up to 64 GB of physical memory
Hal.dll	Hardware abstraction layer
Win32k.sys	Kernel-mode part of the Win32 subsystem
Ntdll.dll	Internal support functions and system service dispatch stubs to executive functions
Kernel32.dll, Advapi32.dll, User32.dll, Gdi32.dll	Core Win32 subsystem DLLs

Debugger Commands: !drivers

- **!drivers** **display systemwide modules**
- operating system executive and kernel (ntoskrnl.exe)
- drivers
- systemwide DLLs (hal.dll, ntdll.dll, etc.)
- others

!drivers

This debugger command displays a list showing each driver and other system-wide loaded on the target computer.

The following example is a partial listing:

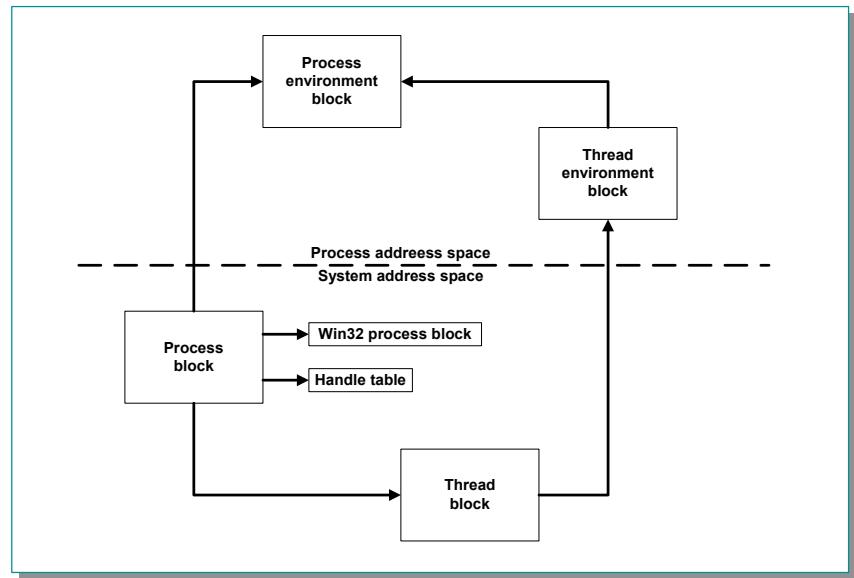
```
kd>!drivers
Loaded System Driver Summary
Base      Code Size      Data Size      Driver Name Creation Time
80080000  f76c0 (989 kb) 1f100 (124 kb) ntoskrnl.exe Fri May 26 15:13:00
80400000  d980   ( 54 kb) 4040   ( 16 kb) hal.dll     Tue May 16 16:50:34
80654000  3f00   ( 15 kb) 1060   (  4 kb) ncrc810.sys Fri May 05 20:07:04
8065a000  a460   ( 41 kb) 1e80   (  7 kb) SCSIPORT.SYS Fri May 05 20:08:05
```

The following information can be determined from the output of the preceding example:

Parameter	Meaning
Base	The starting address of the module code, in hexadecimal. When the memory address used by the code that causes a crash falls between the base address for a module and the base address for the next module in the list, then that module is frequently the cause of the fault. For instance, the base for Nrc810.sys is 0x80654000. Any address between that and 0x8065a000 belongs to this driver.
Code Size	The size in kilobytes of the module code, in both hexadecimal and decimal.
Data Size	The amount of space in kilobytes allocated to the module for data, in both hexadecimal and decimal.
Driver Name	The module filename.
Creation Time	The link date of the module. Do not confuse this with the file date of the module, which can be set by external tools. The link date is set by the compiler when a driver or other executable file is compiled. It should be

	close to the file date, but it is not always the same.
--	--

Processes and Threads



Each process is represented by an executive process (EPROCESS) block. Besides containing many attributes relating to a process, an EPROCESS block contains and points to a number of other related data structures. For example, each process has one or more threads represented by executive thread (ETHREAD) blocks. The EPROCESS block and its related data structures exist in system space, with the exception of the process environment block (PEB), which exists in the process address space (because it contains information that is modified by user-mode code).

In addition to the EPROCESS block, the Win32 subsystem process (Csrss) maintains a parallel structure for each process that executes a Win32 program. Also, the kernel-mode part of the Win32 subsystem (Win32k.sys) has a per-process data structure that is created the first time a thread calls a Win32 USER or GDI function that is implemented in kernel mode.

Debugger Commands: !process

- **!process [proc] [flags]** **display process(es)**
- *proc* is one of:
 - empty or -1 (current process)
 - 0 (all processes)
 - process address
 - process ID (PID)
- *flags* supplies bitmapped options
 - 0 requests minimal information for each process
 - default is 0x7
 - see debugger documentation for other options
- **!process 0 0** brief list of all processes
- **!process proc** default info for one process

!process [Process] [Flags]

Displays summarized or detailed information about the specified process, or about all processes.

Process

Optional, unless you must specify the *Flags* parameter as well. Hexadecimal address of the process on the target computer. A value of -1 (0xFFFFFFFF) indicates the current process. The process ID, or PID, of the process may be used instead of its address. If both *Process* and *Flags* are omitted, the current process is used. If zero is specified, information is displayed for all processes.

Flags

Flags is an optional numeric value indicating the amount of detail to be displayed for the process(es). The default is 0x7, which displays maximal information. See the debugger documentation for additional details.

Here are some commonly used variations of this command:

!process 0 0

Displays brief information for all processes

!process 0 7

Displays maximal information for every process (very long)

!process 0

Same as above

!process Process

Displays maximal information for one process

!process Process 1

Displays complete process information, but no thread information, for one process

Here is some sample output from the latter command, using -1 for *Process* to indicate the current process from a crash dump. This is often important information because it

identifies the program, application, or system process that was executing at the time of the failure. This *may* indicate that operations being performed by that process were involved in the problem.

```
kd> !process -1 1
PROCESS 81286aa0 SessionId: 0 Cid: 00e4 Peb: 7ffdf000 ParentCid: 00c8
    DirBase: 038c2000 ObjectTable: 81288768 TableSize: 719.
    Image: services.exe
    VadRoot 8115d508 Clone 0 Private 1740. Modified 12. Locked 23.
    DeviceMap 81412808
    Token e1b169d0
    ElapsedTime 0:33:19.0025
    UserTime 0:00:02.0894
    KernelTime 0:00:03.0194
    QuotaPoolUsage[PagedPool] 34364
    QuotaPoolUsage[NonPagedPool] 277776
    Working Set Sizes (now,min,max) (2693, 50, 345) (10772KB, 200KB, 1380KB)
    PeakWorkingSetSize 3917
    VirtualSize 60 Mb
    PeakVirtualSize 60 Mb
    PageFaultCount 23810
    MemoryPriority BACKGROUND
    BasePriority 9
    CommitCharge 2416
```

Debugger Commands: !thread

- **!thread [thread] [flags]** **display thread**
- *thread* is one of:
 - empty or -1 (current thread)
 - thread address
- *flags* supplies bitmapped options
 - 0 requests minimal information for each thread
 - default is 0x6 (stack trace and objects being waited on)

!thread [Thread] [Flags]

Displays summarized or detailed information about the specified process, or about all processes.

Thread

Optional, unless *Flags* must be specified. Hexadecimal address of the thread on the target computer. A value of -1 (0xFFFFFFFF) indicates the current thread. If both *Thread* and *Flags* are omitted, the current thread is used.

Flags

Flags is an optional numeric value indicating the amount of detail to be displayed for the selected thread. It can be any of several values; see the debugger documentation for complete details. The default is 0x6, which causes the display to include the thread's stack, objects being waited for, scheduling state, and execution times.

Here are some commonly used variations of this command:

!thread -1 0

Displays brief information for the current thread. This shows the thread address, the thread and process IDs, the thread environment block (TEB) address, the address of the Win32 function (if any) the thread was created to run, and the thread's scheduling state.

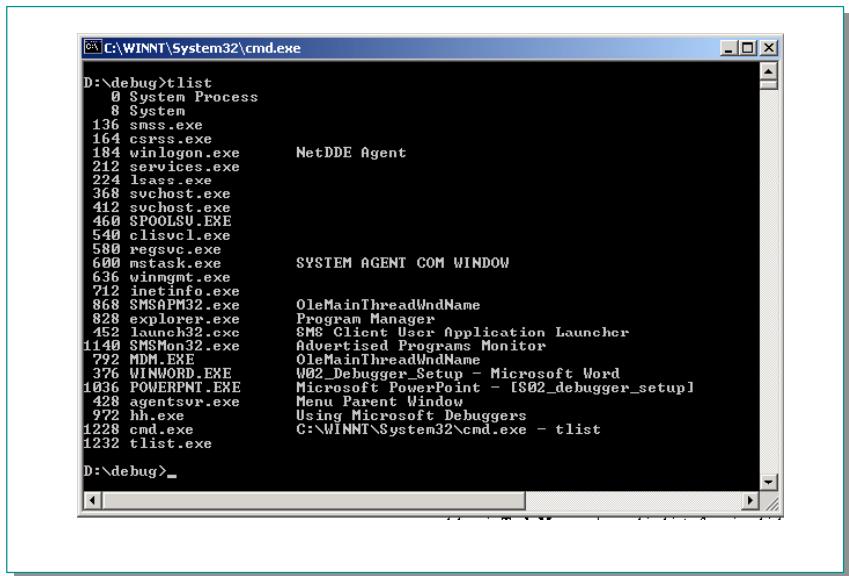
!thread

Displays default information for the current thread. This includes the brief information, the list of I/O requests (if any) that the thread has issued, outstanding

Here we use the command to show brief information for the current thread.

```
kd> !thread -1 0
THREAD 81272020 Cid e4.15c TEB: 7ffda000 Win32Thread: e1b7e888 RUNNING
```

Process ID (PID) Displayed by TList



Finding the Process ID

Each process running in Microsoft® Windows® is assigned a unique decimal number called the *process ID*, or *PID*.

There are several ways to determine the PID for a given application: using the Task Manager, using the **tasklist** command, using the TList utility, or using the debugger.

Task Manager

The *Task Manager* may be activated in a number of ways, but the simplest is to press CTRL+ALT+DELETE and then click on **Task Manager**.

If you select the **Processes** tab, each process and its PID will be listed, along with other useful information.

Some kernel errors may cause delays in **Task Manager**'s graphical interface.

The Tasklist Command

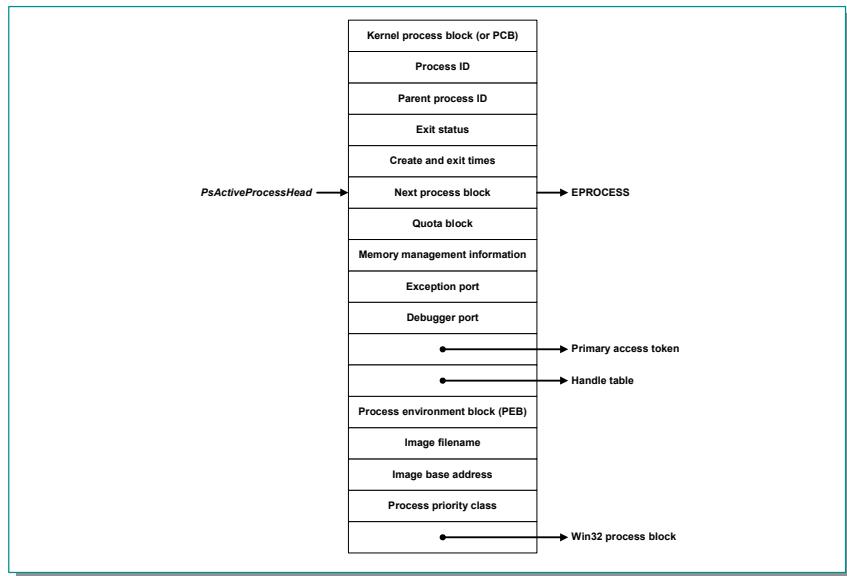
In Windows XP and Windows .NET Server 2003, you can use the **tasklist** command from a Command Prompt window. This displays all processes, their PIDs, and a variety of other details.

TList

TList (*Task List Viewer, tlist.exe*) is a command-line utility that displays a list of tasks, or user-mode processes, currently running on the local computer. TList is included in the Debugging Tools for Windows package.

When you run TList from the command prompt, it will display a list of all the user-mode processes in memory with a unique process identification (PID) number. For each process, it shows the PID, process name, and, if the process has a window, the title of that window.

EPROCESS

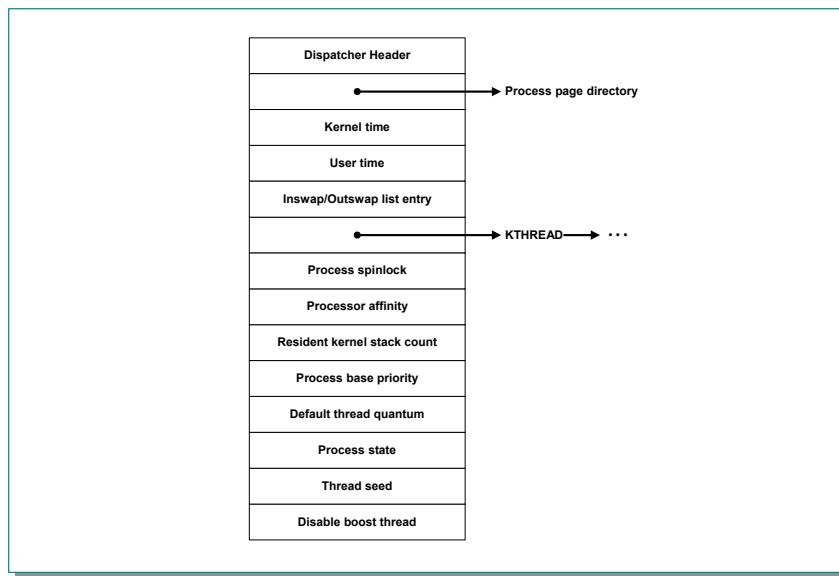


Key Fields in the EPROCESS Block

Element	Purpose
Kernel process (KPROCESS) block	Common dispatcher object header, pointer to the process page directory, list of kernel thread (KTHREAD) blocks belonging to the process, default base priority, quantum, affinity mask, and total kernel and user time for the threads in the process.
Process identification	Unique process ID, creating process ID, name of image being run, window station process is running on.
Quota block	Limits on nonpaged pool, paged pool, and page file usage plus current and peak process nonpaged and paged pool usage. (Note: Several processes can share this structure: all the system processes point to the single systemwide default quota block; all the processes in the interactive session share a single quota block Winlogon sets up.)
Virtual address descriptors (VADs)	Series of data structures that describes the status of the portions of the address space that exist in the process.
Working set information	Pointer to working set list (MMWSL structure); current, peak, minimum, and maximum working set size; last trim time; page fault count; memory priority; outswap flags; page fault history.
Virtual memory information	Current and peak virtual size, page file usage, hardware page table entry for process page directory.

Exception local procedure call (LPC) port	Interprocess communication channel to which the process manager sends a message when one of the process's threads causes an exception.
Debugging LPC port	Interprocess communication channel to which the process manager sends a message when one of the process's threads causes a debug event.
Access token (ACCESS_TOKEN)	Executive object describing the security profile of this process.
Handle table	Address of per-process handle table.
Device map	Address of object directory to resolve device name references in (supports multiple users).
Process environment block (PEB)	Image information (base address, version numbers, module list), process heap information, and thread-local storage utilization. (Note: The pointers to the process heaps start at the first byte after the PEB.)
Win32 subsystem process block (W32PROCESS)	Process details needed by the kernel-mode component of the Win32 subsystem.

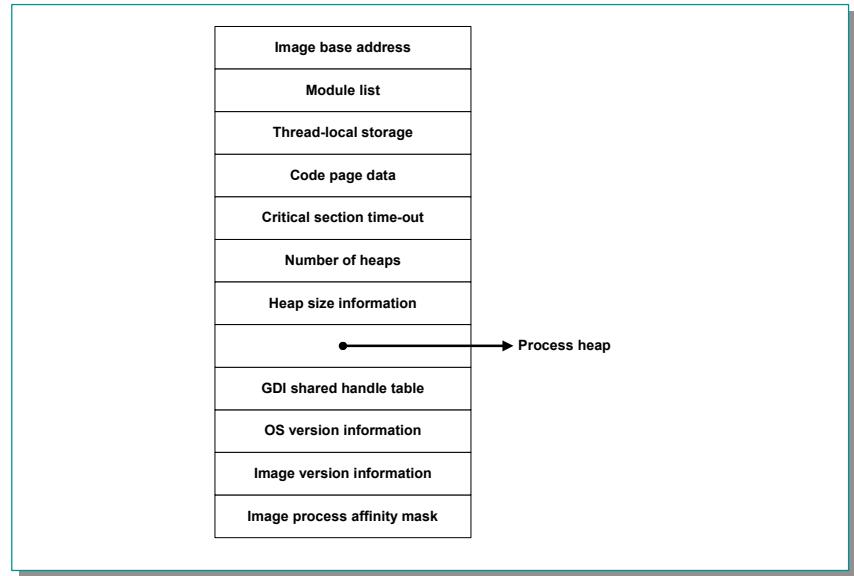
KPROCESS



Kernel Processor Block (PCB)

Two key substructures of the executive process block are the kernel process (KPROCESS) block and the process environment block (PEB). The KPROCESS block (which is sometimes called the PCB, or process control block) is illustrated in this slide and contains the information that the kernel needs to schedule threads.

Process Environment Block - PEB



Process Environment Block

The PEB, which lives in the user process address space, contains information needed by the image loader, the heap manager, and other Win32 system DLLs that need to be writable from user mode. (The EPROCESS and KPROCESS blocks are accessible only from kernel mode.) The PEB is always mapped at address 0x7FFDF000 on x86 platforms.

Debugger Commands: !processfields and DT

- **!processfields** shows structure of EPROCESS
(Windows 2000 and earlier only)
- **dt nt!_EPROCESS** dt = display type
shows structure of EPROCESS
(Windows XP and later only)
- **dt nt!_EPROCESS processAddress**
shows structure of EPROCESS with
values taken from indicated process
- **dt nt!_KPROCESS** similar to above, for KPROCESS

!processfields

For Windows 2000 target systems, this command lets you examine the details of the EPROCESS structure.

```
kd> !processfields
EPROCESS structure offsets:

Pcb: 0x0
ExitStatus: 0x6c
LockEvent: 0x70
LockCount: 0x80
CreateTime: 0x88
ExitTime: 0x90
LockOwner: 0x98
UniqueProcessId: 0x9c
ActiveProcessLinks: 0xa0
QuotaPeakPoolUsage[0]: 0xa8
QuotaPoolUsage[0]: 0xb0
PagefileUsage: 0xb8
CommitCharge: 0xbc
PeakPagefileUsage: 0xc0
PeakVirtualSize: 0xc4
VirtualSize: 0xc8
Vm: 0xd0
DebugPort: 0x120
ExceptionPort: 0x124
ObjectTable: 0x128
Token: 0x12c
WorkingSetLock: 0x130
WorkingSetPage: 0x150
ProcessOutswapEnabled: 0x154
ProcessOutswapped: 0x155
[...]
```

For Windows XP and later target systems, the **dt, Display Type**, command may be used instead to show both the EPROCESS and KPROCESS structures. **dt** takes as its first parameter the name of a data type and displays a description of that type. The requested type must be included in the symbol files available to the debugger. Normally, public symbol files do not include type information, but type information is included in the

Windows XP and later public symbol files for many of the more important data structures.

In general the “structure type” name form (with a leading underscore) should be used with this command, although typedef names (without the underscore) can usually be processed as well.

The “nt!” prefix in the examples instructs the debugger to look only in the nt (short for ntoskrnl) module for this type. This saves time that the debugger might also have to spend searching the symbol files for all loaded modules to find the definition of the specified type. Here is an example of **dt** used on an XP target system to show the layout of the KPROCESS structure:

```
0: kd> dt nt!_KPROCESS
+0x000 Header : _DISPATCHER_HEADER
+0x010 ProfileListHead : _LIST_ENTRY
+0x018 DirectoryTableBase : [2] Uint4B
+0x020 LdtDescriptor : _KGDTENTRY
+0x028 Int21Descriptor : _KIDTENTRY
+0x030 IopmOffset : Uint2B
+0x032 Iopl : UChar
+0x033 Unused : UChar
+0x034 ActiveProcessors : Uint4B
+0x038 KernelTime : Uint4B
+0x03c UserTime : Uint4B
+0x040 ReadyListHead : _LIST_ENTRY
+0x048 SwapListEntry : _SINGLE_LIST_ENTRY
+0x04c VdmTrapcHandler : Ptr32 Void
+0x050 ThreadListHead : _LIST_ENTRY
+0x058 ProcessLock : Uint4B
+0x05c Affinity : Uint4B
+0x060 StackCount : Uint2B
+0x062 BasePriority : Char
+0x063 ThreadQuantum : Char
+0x064 AutoAlignment : UChar
+0x065 State : UChar
+0x066 ThreadSeed : UChar
+0x067 DisableBoost : UChar
+0x068 PowerState : UChar
+0x069 DisableQuantum : UChar
+0x06a IdealNode : UChar
+0x06b Spare : UChar
```

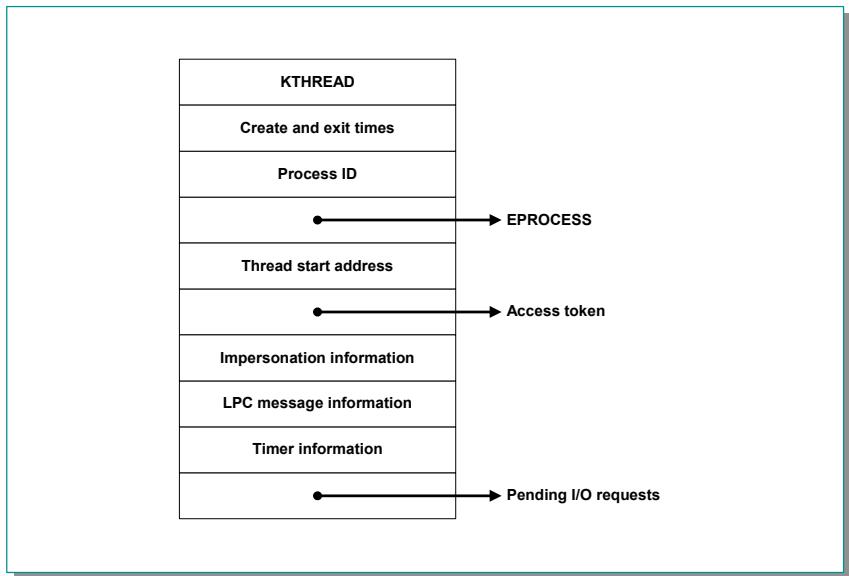
If you have the address of a process, you can use this as a second parameter to **dt** for the _KPROCESS or _EPROCESS structures, to show the field offsets along with the field contents for the indicated instance of the structure. Here, for example, we use !process to obtain the address of these structures for the current process:

```
0: kd> !process -1 0
PROCESS 821eb030 SessionId: 0 Cid: 01e8 Peb: 7ffdf000 ParentCid: 01a8
DirBase: 0fc66000 ObjectTable: e177e178 TableSize: 567.
Image: csrss.exe
```

Then we use the **dt** command to format the _KPROCESS structure for this process:

```
0: kd> dt nt!_KPROCESS 821eb030
+0x000 Header          : _DISPATCHER_HEADER
+0x010 ProfileListHead : _LIST_ENTRY [ 0x821eb040 - 0x821eb040 ]
+0x018 DirectoryTableBase : [2] 0xfc66000
+0x020 LdtDescriptor   : _KGDTENTRY
+0x028 Int21Descriptor : _KIDTENTRY
+0x030 IopmOffset     : 0x20ac
+0x032 Iopl           : 0x1 ''
+0x033 Unused          : 0 ''
+0x034 ActiveProcessors : 1
+0x038 KernelTime      : 0x54da
+0x03c UserTime         : 0x1c53
+0x040 ReadyListHead    : _LIST_ENTRY [ 0x821eb070 - 0x821eb070 ]
+0x048 SwapListEntry    : _SINGLE_LIST_ENTRY
+0x04c VdmTrapcHandler : (null)
+0x050 ThreadListHead   : _LIST_ENTRY [ 0x820db3b0 - 0x820d41e0 ]
+0x058 ProcessLock      : 0
+0x05c Affinity          : 3
+0x060 StackCount        : 5
+0x062 BasePriority      : 13 ''
+0x063 ThreadQuantum    : 6 ''
+0x064 AutoAlignment     : 0 ''
+0x065 State             : 0 ''
+0x066 ThreadSeed        : 0 ''
+0x067 DisableBoost       : 0 ''
+0x068 PowerState         : 0 ''
+0x069 DisableQuantum    : 0 ''
+0x06a IdealNode          : 0 ''
+0x06b Spare              : 0 ''
```

ETHREAD

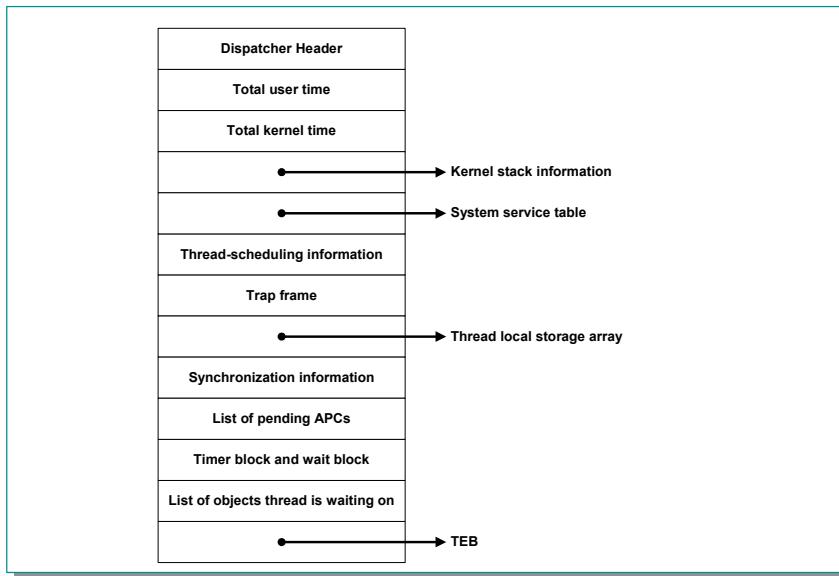


At the operating system level, a thread is represented by an executive thread (ETHREAD) block. The ETHREAD block and the structures it points to exist in the system address space, with the exception of the thread environment block (TEB), which exists in the process address space. In addition, the Win32 subsystem process (Csrss) maintains a parallel structure for each thread created in a Win32 process. Also, for threads that have called a Win32 subsystem USER or GDI function, the kernel-mode portion of the Win32 subsystem (Win32k.sys) maintains a per-thread data structure (called the W32THREAD structure) that the ETHREAD block points to.

Key fields in the Executive Thread Block

Element	Description
KTHREAD block	Pointer to KTHREAD data structure
Thread time information	Thread create and exit time
Process identification	Process ID and pointer to EPROCESS block of the process that the thread belongs to
Start address	Address of thread start routine
Impersonation information	Access token and impersonation level (if the thread is impersonating a client)
LPC information	Message ID that the thread is waiting for and address of message
I/O information	List of pending I/O request packets (IRPs)

KTHREAD



KTHREAD

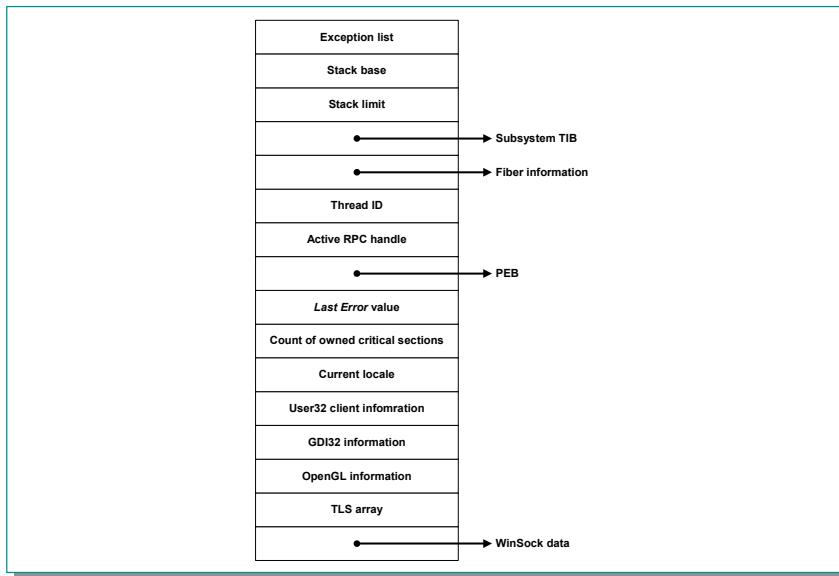
The KTHREAD block contains the information that the kernel needs to access to perform thread scheduling and synchronization on behalf of running threads.

Key Contents of the KTHREAD Block

Element	Description
Dispatcher header	Because the thread is an object that can be waited on, it starts with a standard kernel dispatcher object header.
Execution time	Total user and kernel CPU time.
Pointer to kernel stack information	Base and upper address of the kernel stack.
Pointer to system service table	Each thread starts out with this field pointing to the main system service table (KeServiceDescriptorTable). When a thread first calls a Win32 GUI service, its system service table is changed to one that includes the GDI and USER services in Win32k.sys.
Scheduling information	Base and current priority, quantum, affinity mask, ideal processor, scheduling state, freeze count, and suspend count.
Wait blocks	The thread block contains four built-in wait blocks so that wait blocks don't have to be allocated and initialized each time the thread waits on something. (One wait block is dedicated to timers.)

Wait information	List of objects the thread is waiting on, wait reason, and time at which the thread entered the wait state.
Mutant list	List of mutant objects the thread owns.
APC queues	List of pending user-mode and kernel-mode APCs, and alertable flag.
Timer block	Built-in timer block (also a corresponding wait block).
Queue list	Pointer to queue object that the thread is associated with.
Pointer to TEB	Thread ID, TLS information, PEB pointer, and GDI and OpenGL information.

Thread Environment Block - TEB



Thread Environment Block

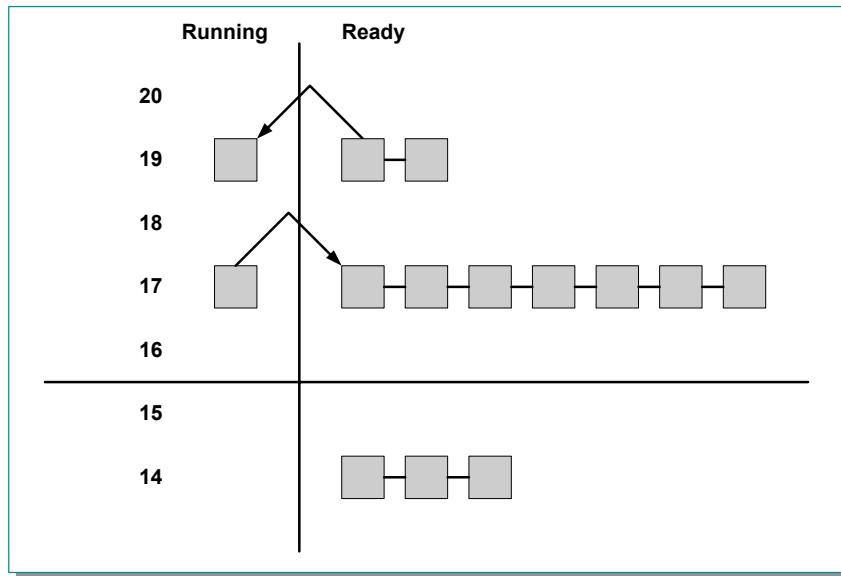
The TEB stores context information for the image loader and various Win32 DLLs. Because these components run in user mode, they need a data structure writable from user mode. That is why this structure exists in the user address space instead of in the system space, where it would be writable only from kernel mode. You can find the address of the TEB with the kernel debugger `!thread` command.

Debugger Commands: !threadfields and DT

- **!threadfields** shows structure of ETHREAD
(Windows 2000 and earlier only)
- **dt nt!_ETHREAD** dt = display type
shows structure of ETHREAD
(Windows XP and later only)
- **dt nt!_ETHREAD processAddress** shows structure of ETHREAD with
values taken from indicated thread
- **dt nt!_KTHREAD** similar to above, for KTHREAD

The **!threadfields**, and **dt** command for the _ETHREAD and _KTHREAD structures, operate in a similar fashion to their counterparts for the process structures.

Thread Scheduling



Windows implements a priority-driven, preemptive scheduling system—the highest-priority runnable (*ready*) thread always runs, with the caveat that the thread chosen to run might be limited by the processors on which the thread is allowed to run, a phenomenon called *processor affinity*. By default, threads can run on any available processor, but you can alter processor affinity by using one of the Win32 scheduling functions.

When a thread is selected to run, it runs for an amount of time called a *quantum*. A quantum is the length of time a thread is allowed to run before Windows interrupts the thread to find out whether another thread at the same priority level is waiting to run or whether the thread's priority needs to be reduced. Quantum values can vary from thread to thread (and between different builds of Windows). A thread might not get to complete its quantum, however. Because Windows implements a preemptive scheduler, if another thread with a higher priority becomes ready to run, the currently running thread might be preempted before finishing its time slice. In fact, a thread can be selected to run next and be preempted before even beginning its quantum!

The Windows scheduling code is implemented in the kernel. There's no separate "scheduler" thread, or even a single routine. Scheduling decisions are made in two different top-level routines. These are called from any other routines in the kernel in which scheduling-related events occur. The routines that perform these duties are collectively called the kernel's *dispatcher*. Thread dispatching occurs at DPC/Dispatch level and is triggered by any of the following events:

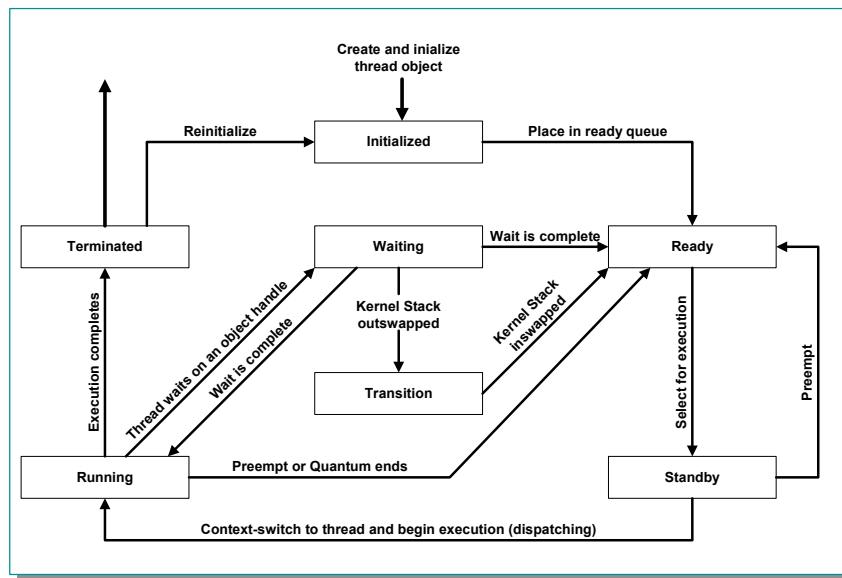
- A thread becomes ready to execute—for example, a thread has been newly created or has just been released from the wait state.
- A thread leaves the running state because its time quantum ends, it terminates, or it enters a wait state.
- A thread's priority changes, either because of a system service call or because Windows itself changes the priority value.
- The processor affinity of a running thread changes.

When any of these events occur, Windows must determine which thread should run next. When Windows selects a new thread to run, it performs a context switch to it. A context

switch is the procedure of saving the volatile machine state associated with a running thread, loading another thread's volatile state, and starting the new thread's execution.

As already noted, scheduling is done to threads, not to entire processes. This approach makes sense when you consider that processes don't run but only provide resources and a context in which their threads run. Because scheduling decisions are made strictly on a thread basis, no consideration is given to what process the thread belongs to. For example, if process **A** has 10 runnable threads and process **B** has 2 runnable threads, and all 12 threads are at the same priority, each thread would receive one-twelfth of the CPU time—Windows wouldn't give 50 percent of the CPU to process A and 50 percent to process B.

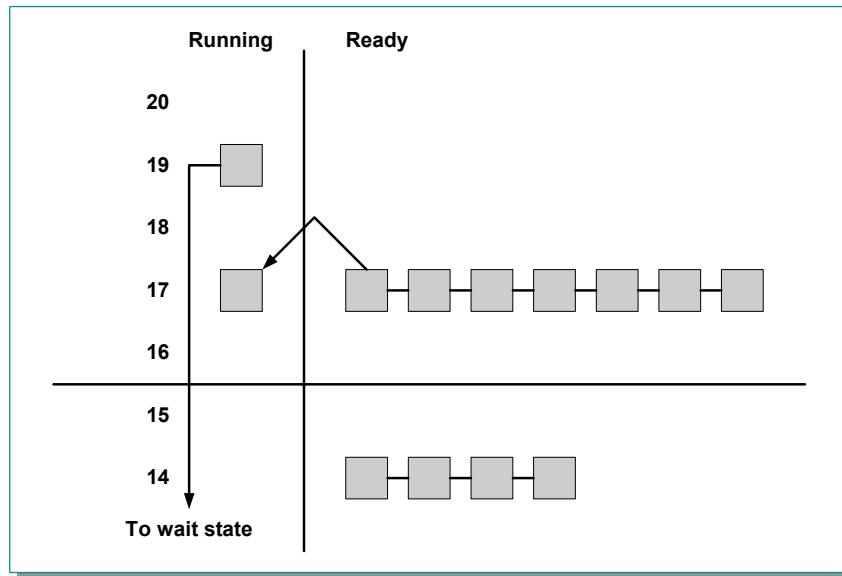
Thread States



The following are the possible thread state values:

- **Ready** When looking for a thread to execute, the dispatcher considers only the pool of threads in the ready state. These threads are simply waiting to execute.
- **Standby** A thread in the standby state has been selected to run next on a particular processor. When the correct conditions exist, the dispatcher performs a context switch to this thread. Only one thread can be in the standby state for each processor on the system.
- **Running** Once the dispatcher performs a context switch to a thread, the thread enters the running state and executes. The thread's execution continues until the kernel preempts it to run a higher priority thread, its quantum ends, it terminates, or it voluntarily enters the wait state.
- **Waiting** A thread can enter the wait state in several ways: a thread can voluntarily wait on an object to synchronize its execution, the operating system (the I/O system, for example) can wait on the thread's behalf, or an environment subsystem can direct the thread to suspend itself. When the thread's wait ends, depending on the priority, the thread either begins running immediately or is moved back to the ready state.
- **Transition** A thread enters the transition state if it is ready for execution but its kernel stack is paged out of memory. For example, the thread's kernel stack might be paged out of memory. Once its kernel stack is brought back into memory, the thread enters the ready state.
- **Terminated** When a thread finishes executing, it enters the terminated state. Once terminated, a thread object might or might not be deleted. (The object manager sets policy regarding when to delete the object.) If the executive has a pointer to the thread object, it can reinitialize the thread object and use it again.
- **Initialized** Used internally while a thread is being created.

Voluntary Switching



Scenarios

Windows bases the question of “Who gets the CPU?” on thread priority; but how does this work in practice? The following sections illustrate just how priority-driven preemptive multitasking works on the thread level.

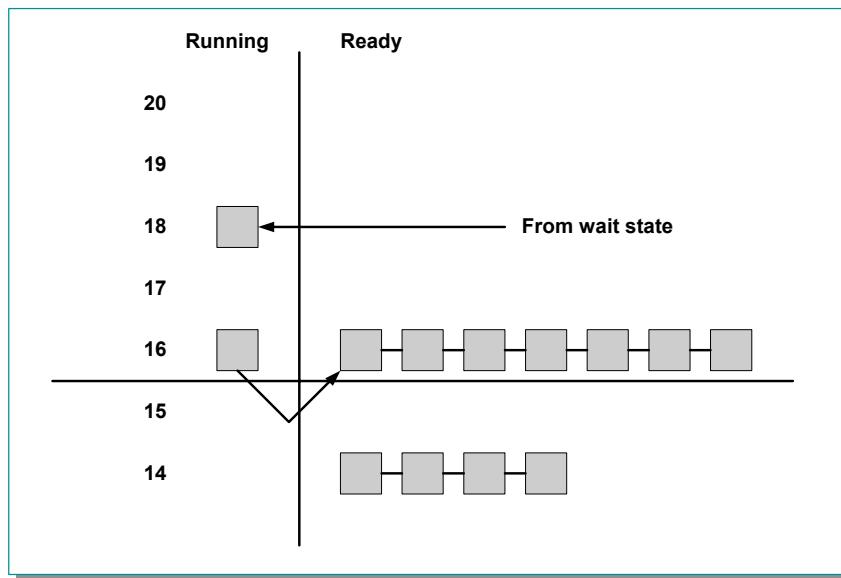
Voluntary Switch

First a thread might voluntarily relinquish use of the processor by entering a wait state on some object (such as an event, mutex, semaphore, I/O completion port, process, thread, window message, and so on) by calling one of the many Win32 wait functions (such as *WaitForSingleObject* or *WaitForMultiple Objects*).

Voluntary switching is roughly equivalent to a thread ordering an item that isn’t ready to go at a fast-food counter. Rather than hold up the queue of the other diners, the thread will step aside and let the next thread execute its routine while the first thread’s hamburger is being prepared. When the hamburger is ready, the first thread goes to the end of ready queue of the priority level. However, most wait operations result in a temporary priority boost so that the thread can pick up its hamburger right away and start eating.

In this slide, the top block (thread) is voluntarily relinquishing the processor so that the next thread in the ready queue can run. Although it might appear from this slide that the relinquishing thread’s priority is being reduced, it’s not—it’s just being moved to the wait queue of the objects(s) the thread is waiting on. What about any remaining quantum for the thread? The quantum value is not reset when a thread enters a wait state—in fact, when the wait is satisfied, the thread’s quantum value is decremented by 1 quantum unit, equivalent to one third of a clock interval.

Preemption



In this scheduling scenario, a lower-priority thread is preempted when a higher-priority thread becomes ready to run. This situation might occur for a couple of reasons.

- A higher-priority thread's wait completes. (The event that the other thread was waiting on has occurred.)
- A thread priority is increased above that of the running thread.
- The running thread's priority is decreased below that of a ready thread.

In each case, the running thread will be preempted to allow a higher-priority thread to run.

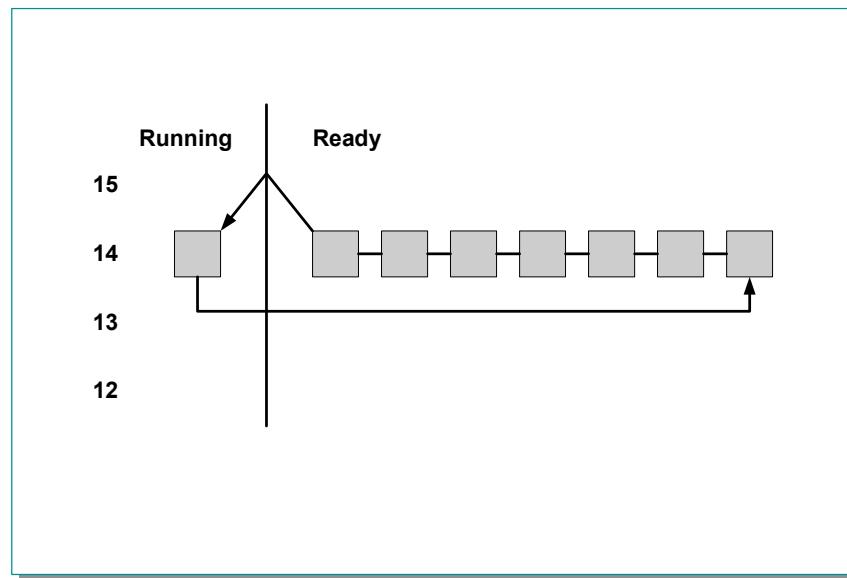
NOTE Threads running in kernel mode can be preempted by threads running in user mode—the mode in which the thread is running doesn't matter. The thread priority is the determining factor.

When a thread is preempted, it is put at the head of the ready queue for the priority it was running at so that it can finish its quantum when it gets to run again. Although the thread won't get to restart its time slice, it will get to complete any time remaining in its quantum.

In this slide, a thread with priority 18 emerges from a wait state and repossesses the CPU, causing the thread that had been running (at priority 16) to be bumped to the head of the ready queue. Notice that the bumped thread is not going to the end of the queue but to the beginning; when the preempting thread has finished running, the bumped thread can complete its quantum. In this example, the threads are in the real-time range; no dynamic priority boosts are allowed for threads in the real-time range.

If voluntary switching is roughly equivalent to a thread letting another thread place its lunch order while the first thread waits for its meal, preemption is roughly equivalent to a thread being bumped from its place in line because the president of the United States has just walked in and ordered a hamburger. The preempted thread doesn't get bumped to the back of the line but is simply moved aside while the president gets his lunch. As soon as the president leaves, the first thread can resume ordering its meal.

Quantum End



Quantum

A quantum is the amount of time a thread gets to run before Windows checks whether another thread at the same priority should get to run. If a thread completes its quantum and there are no other threads at its priority, Windows reschedules the thread to run for another quantum.

Each thread has a quantum value that represents how long the thread can run until its quantum expires. This value isn't a time length but rather an integer value, which we'll call *quantum units*.

Quantum Accounting

By default, threads start with a quantum value of 6 on Windows Professional and 36 on Windows Server. The rationale for the longer default value on Windows Server is to minimize context switching. By having a longer quantum, server applications that wake up as the result of a client request have a better chance of completing the request and going back into a wait state before their quantum ends.

Each time the clock interrupt, the clock-interrupt routine deducts a fixed value (3) from the thread quantum. If there is no remaining thread quantum, the quantum end processing is triggered and another thread might be selected to run. On Windows Professional, because 3 is deducted each time the clock interrupt fires, by default a thread runs for 2 clock intervals; on Windows Server, by default a thread runs for 12 clock intervals.

Even if the system were at DPC/dispatch level or above (for example, if a DPC or an interrupt service routine was executing) when the clock interrupt occurred, the current thread would still have its quantum decremented, even if it hadn't been running for a full clock interval. If this was not done and device interrupts or DPCs occurred right before the clock interval timer interrupts, threads might not ever get their quantum reduced.

The length of the clock interval varies according to the hardware platform. The frequency of the clock interrupts is up to the HAL, not the kernel. For example, the clock interval for most x86 uniprocessors is 10 milliseconds, and for most x86 multiprocessors, 15 milliseconds.

The reason quantum is expressed in terms of a multiple of 3 quantum units per clock tick rather than as single units is to allow for partial quantum decay on wait completion. When a thread that has a base priority less than 14 executes a wait function (such as *WaitForSingleObject* or *WaitForMultipleObjects*), its quantum is reduced by 1 quantum unit. (Threads running at priority 14 or higher have their quantums reset after a wait.)

This partial decay addresses the case in which a thread enters a wait state before the clock interval timer fires. If this adjustment were not made, it would be possible for threads never to have their quantums reduced. For example, if a thread ran, entered a wait state, ran again, and entered another wait state but was never the currently running thread when the clock interval timer fired, it would never have its quantum charged for the time it was running.

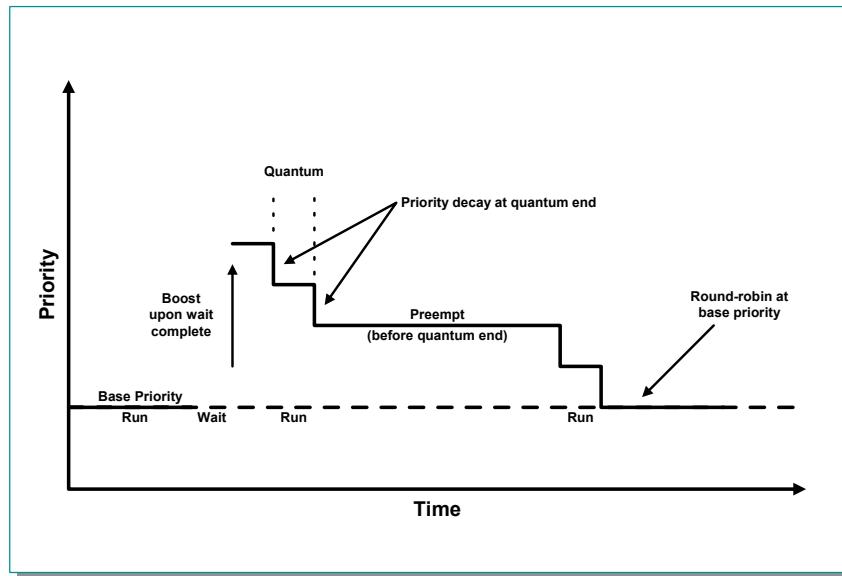
Quantum End

When the running thread exhausts its CPU quantum, Windows must determine whether the thread's priority should be decremented and then whether another thread should be scheduled on the processor.

If the thread priority is reduced, Windows looks for a more appropriate thread to schedule. (For example, a more appropriate thread would be a thread in a ready queue with a higher priority than the new priority for the currently running thread.) If the thread priority is not reduced, Windows selects the next thread in the ready queue at the same priority level and moves the previously running thread to the tail of that queue (giving it a new quantum value and changing its state from running to ready). This case is illustrated in this slide. If no other thread of the same priority is ready to run, the thread gets to run for another quantum.

The fact that a thread has a quantum does not mean that it must finish the quantum. A thread might voluntarily relinquish control of the CPU before its time slice ends by going into a wait state, or it might be preempted before finishing its time slice by a thread with a higher priority. In either case, when the thread is allowed to resume executing it begins with whatever quantum it had remaining. (An exception is made for threads of priority 16 or above – their quantum is reset upon preemption, so they begin with a full quantum each time they run.)

Priority Boost and Decay



Priority Adjustments

The priorities of threads are not fixed. Threads are assigned a base priority upon creation, but if the base priority of a thread is below 15, its actual, or current, priority can be boosted as high as 15, depending on what the thread is doing. Its priority may then decay back down to the base, again depending on what the thread is doing while it's running with a boosted priority.

Most priority boosts occur as a side effect of *wait resolution*. That is, a thread that has been waiting, or blocked, for some reason gets a priority boost when its wait is resolved.

Boosts are also given to threads if they have been Ready, that is, in a Ready queue but not executing, for more than a few seconds. This is a “starvation avoidance” mechanism to ensure that all threads, even those of low priority, get to run occasionally for short times.

Once a thread is running at a boosted priority, its priority will then be decayed, or reduced, whenever the thread experiences quantum end. Its priority will never decay below the base priority for the thread.

The result is that I/O bound threads tend to stay at boosted priorities, while compute-bound threads stay at their base. The intent is to give rapid access to the CPU to threads that are performing I/O operations, so that they can initiate the next such operation quickly, thereby improving I/O throughput.

Automatic priority adjustments can be disabled on a per-thread basis. They are also not applied at all to threads whose base priority is 16 or above (the so-called “real time” range). The relative priorities of threads with base priorities in the range 16 and above are therefore predictable, while those of threads with base priorities 15 and below are not.

Debugger Commands: !thread, !pcr, !ready, !locks

- **!thread** shows current *Running* thread
- **!pcr 0, 1, ...** shows Processor Control Region – including Current (*Running*) and Next thread for CPU 0, 1, etc
- **!ready** lists all *Ready* threads
- **!thread *threadAddress***
shows thread state
for WAITing thread, lists objects being waited for
- **!locks** displays kernel “executive resource” locks (threads can be waiting for these)

Looking at Threads

There are normally a great many threads in a computer running the Windows operating system. In a debugging situation it is important to be able to identify the ones that are most likely involved in the problem being examined. The debugger provides several ways to do this.

!thread

As described previously, this command with either a missing first parameter, or a first parameter of 0xFFFFFFFF, will display the current thread.

```
0: kd> !thread 0xffffffff 1
THREAD 820aada8 Cid 1e8.214 Teb: 7ffd9000 Win32Thread: e1e989b0
RUNNING on processor 0
```

!pcr *processorNumber*

The debugger’s **!pcr** command results in a display of the *Processor Control Region* for the indicated processor. For a multiprocessor target, this can help identify the currently running thread on each processor. Other important information is in the PCR as well.

For example, the previous **!thread** command told us which thread was running on processor 0. But the thread running on processor 1 (if any) might be involved in the crash as well. For purposes of illustration we show the **!pcr** output for both CPUs. Note that the “CurrentThread” address for CPU 0 matches that shown in the above **!thread** output for the current thread:

```

0: kd> !pcr 0
KPCR for Processor 0 at ffdfff000:
Major 1 Minor 1
    NtTib.ExceptionList: f5826218
        NtTib.StackBase: f5828df0
        NtTib.StackLimit: f5826000
        NtTib.SubSystemTib: 00000000
            NtTib.Version: 00000000
            NtTib.UserPointer: 00000000
            NtTib.SelfTib: 7ffd9000

                SelfPcr: ffdfff000
                    Prcb: ffdfff120
                    Irql: 00000000
                    IRR: 00000000
                    IDR: ffffffff
                    InterruptMode: 00000000
                    IDT: 8003f400
                    GDT: 8003f000
                    TSS: 80539700

                    CurrentThread: 820aada8
                    NextThread: 00000000
                    IdleThread: 80543960

                    DpcQueue:

0: kd> !pcr 1
KPCR for Processor 1 at f87d6000:
Major 1 Minor 1
    NtTib.ExceptionList: ffffffff
        NtTib.StackBase: f2b91df0
        NtTib.StackLimit: f2b8f000
        NtTib.SubSystemTib: 00000000
            NtTib.Version: 00000000
            NtTib.UserPointer: 00000000
            NtTib.SelfTib: 7ffdd000

                SelfPcr: f87d6000
                    Prcb: f87d6120
                    Irql: 00000000
                    IRR: 00000000
                    IDR: ffffffff
                    InterruptMode: 00000000
                    IDT: f87da560
                    GDT: f87da160
                    TSS: f87d6d70

                    CurrentThread: ff5f78e0
                    NextThread: 00000000
                    IdleThread: f87d8e20

                    DpcQueue: 0

```

We can then use the **!thread** command with the current thread address from CPU 1, to find out what was running on that CPU at the time of the crash:

```

0: kd> !thread ff5f78e0
THREAD ff5f78e0 Cid 354.448 Peb: 7fffd000 Win32Thread: 00000000
RUNNING on processor 1
Not impersonating
GetULONGFromAddress: unable to read from 00000000
Owning Process ff5fc90
WaitTime (ticks) 830193
Context Switch Count 7312506
UserTime 2:56:45.0343
KernelTime 0:02:17.0984
Start Address 0x77e802f4
Win32 Start Address 0x00422752
Stack Init f2b92000 Current f2b91d34 Base f2b92000 Limit f2b8f000 Call 0
Priority 2 BasePriority 2 PriorityDecrement 0 DecrementCount 16
ChildEBP RetAddr Args to Child
00caf10 00000000 00000000 00000000 00000000 0x4087d9

```

Finally, the **!process** command, used on the “owning process” address from the previous display, will tell us what program was running the thread on CPU 1:

```

0: kd> !process ff5fc90 1
PROCESS ff5fc90 SessionId: 0 Cid: 0354 Peb: 7ffdf000 ParentCid: 02c0
DirBase: 1e483000 ObjectTable: e24cd418 TableSize: 28.
Image: SETI@home.exe
VadRoot ff769508 Vads 60 Clone 0 Private 3614. Modified 120. Locked 0.
DeviceMap e2570998
Token e2cc34f8
ElapsedTime 3:34:09.0656
UserTime 2:56:45.0828
KernelTime 0:02:18.0984
QuotaPoolUsage[PagedPool] 27264
QuotaPoolUsage[NonPagedPool] 2400
Working Set Sizes (now,min,max) (3453, 50, 345) (13812KB, 200KB,
1380KB)
PeakWorkingSetSize 4313
VirtualSize 42 Mb
PeakVirtualSize 43 Mb
PageFaultCount 2037638
MemoryPriority BACKGROUND
BasePriority 4
CommitCharge 3972

```

In this case CPU 1 was running a thread belonging to “SETI@home.exe,” a distributed computing application. This application runs whenever a CPU is free, and therefore probably was not involved in the crash.

!ready

Also of interest might be threads that are trying to run – that is, they are in the *Ready* scheduling state – but can’t because no CPU is available for them. The **!ready** command will provide a list of these threads. **!ready** takes a *Flags* argument much like **!thread** specifying different amounts of detail in the output; following is one of the brief forms:

```

kd> !ready 1
Ready Threads at priority 10
    THREAD 81110020 Cid 4bc.2cc Teb: 7ffde000 Win32Thread: e1e6a168 READY
Ready Threads at priority 9
    THREAD 8104b4e0 Cid 404.200 Teb: 7ffde000 Win32Thread: e1db2448 READY
    THREAD 8104b020 Cid 1dc.3ec Teb: 7ffde000 Win32Thread: e13258e8 READY
    THREAD 81128a40 Cid e4.528 Teb: 7ff99000 Win32Thread: 00000000 READY
Ready Threads at priority 8
    THREAD 810d08a0 Cid 470.440 Teb: 7ffdb000 Win32Thread: e1e95008 READY
    THREAD 81200620 Cid 210.250 Teb: 7ffa0000 Win32Thread: 00000000 READY
    THREAD 811e53e0 Cid 210.218 Teb: 7ffd9000 Win32Thread: 00000000 READY
    THREAD 811b1da0 Cid 310.39c Teb: 7fffaa000 Win32Thread: 00000000 READY
    THREAD 810d1980 Cid 4bc.350 Teb: 7ffd7000 Win32Thread: e1bd5c88 READY
    THREAD 8111cc40 Cid 508.4c8 Teb: 7ffdd000 Win32Thread: e1cb48a8 READY
    THREAD 810a04a0 Cid 1dc.48c Teb: 7ffdd000 Win32Thread: elfa7568 READY
Ready Threads at priority 0
    THREAD 81412020 Cid 8.4 Teb: 00000000 Win32Thread: 00000000 READY
You can then use the !thread and !process commands to further examine these threads
and their owning processes.

```

!thread

If a thread is expected to run, but doesn't, the reason may be that it is in the *Waiting* state, waiting for one or more synchronization objects. The **!thread** command can provide a list of the objects a thread is waiting for.

!locks

Executive resource locks, also known as ERESOURCE locks, are a special synchronization object used internally by the operating system, particularly in the area of file system drivers. The **!locks** extension displays all locks held on resources by threads. A lock can be shared or exclusive, which means no other threads can gain access to that resource. This information is useful when a deadlock occurs on a system. A deadlock is caused by one non-executing thread holding an exclusive lock on a resource needed by an executing thread.

You can usually pinpoint a deadlock in Windows by finding one non-executing thread that holds an exclusive lock on a resource needed by an executing thread. Most of the locks are shared locks.

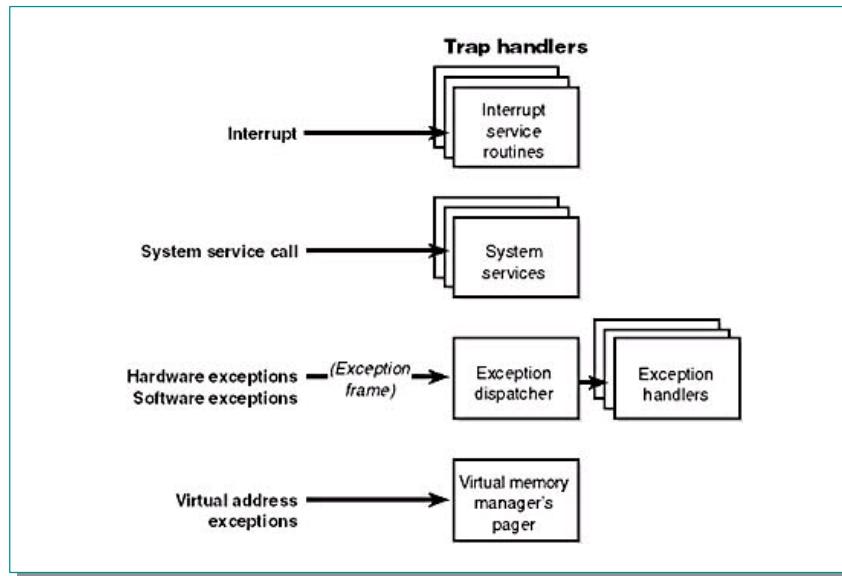
Examples:

```

kd> !locks
***** DUMP OF ALL RESOURCE OBJECTS *****
KD: Scanning for held locks.................
Resource @ 0x81785ca0 Shared 1 owning threads
    Threads: 809daca0-01
Resource @ 0x808728c0 Shared 1 owning threads
    Threads: 809ddcb0-01

```

Trap Dispatching



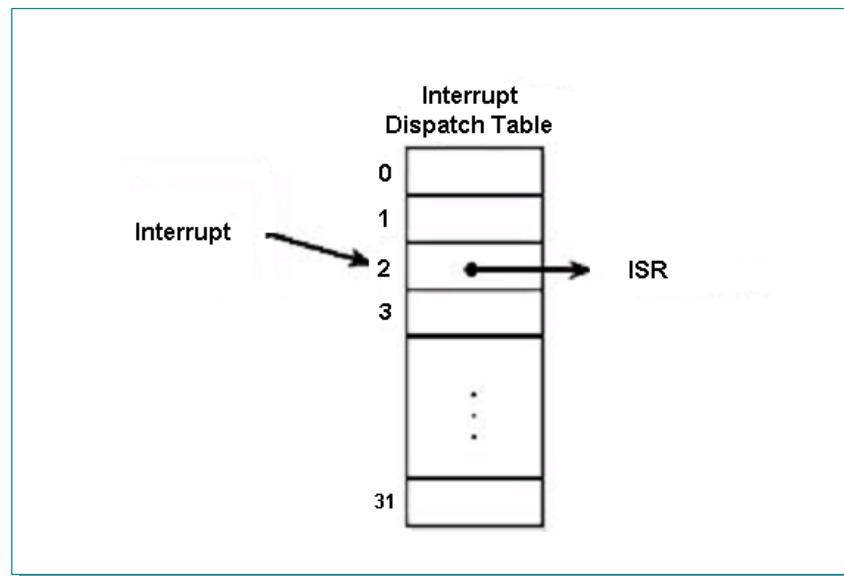
Interrupts and exceptions are conditions that divert the processor to code outside the normal flow of control. Either hardware or software can cause them. The term *trap* refers to a processor's mechanism for breaking into an executing thread when an exception or an interrupt occurs and transferring control to a fixed location in the operating system. In Windows, the processor transfers control to a *trap handler*, a function specific to a particular interrupt or exception.

The kernel distinguishes between interrupts and exceptions in the following way. An *interrupt* is an asynchronous event (one that can occur at any time) that is unrelated to what the processor is executing. Interrupts are generated primarily by I/O devices, processor clocks, or timers, and they can be enabled (turned on) or disabled (turned off). An *exception*, in contrast, is a synchronous condition that results from the execution of a particular instruction. Running a program a second time with the same data under the same conditions can reproduce exceptions. Examples of exceptions include memory access violations, certain debugger instructions, and divide-by-zero errors. The kernel also regards system service calls as exceptions (although technically they're system traps).

Either hardware or software can generate exceptions and interrupts. For example, a bus error exception is caused by a hardware problem, whereas a divide-by-zero exception is the result of a software bug. Likewise, an I/O device can generate an interrupt, or the kernel itself can issue a software interrupt (such as an APC or DPC).

When a hardware exception or interrupt is generated, the processor records enough machine state so that it can return to that point in the control flow and continue execution as if nothing had happened. To do this, the processor creates a *trap frame* on the kernel stack of the interrupted thread into which it stores the execution state of the thread. The trap frame is usually a subset of a thread's complete context. The kernel handles software interrupts either as part of hardware interrupt handling or synchronously when a thread invokes kernel functions related to the software interrupt.

Interrupt Dispatching



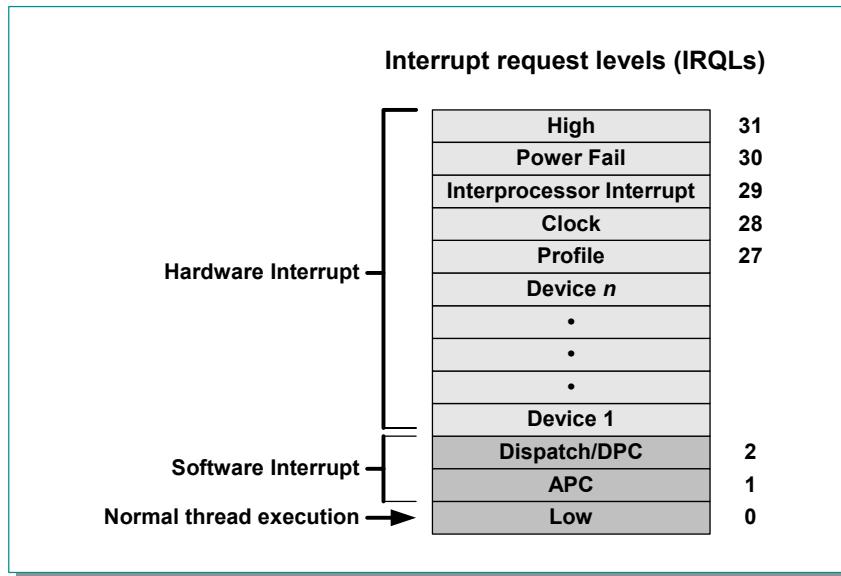
Hardware-generated interrupts typically originate from I/O devices that must notify the processor when they need service. Interrupt-driven devices allow the operating system to get the maximum use out of the processor by overlapping central processing with I/O operations. A thread starts an I/O transfer to or from a device and then can execute other useful work while the device completes the transfer. When the device is finished, it interrupts the processor for service. Pointing devices, printers, keyboards, disk drives, and network cards are generally interrupt driven.

System software can also generate interrupts. For example, the kernel can issue a software interrupt to initiate thread dispatching and to asynchronously break into the execution of a thread. The kernel can also disable interrupts so that the processor isn't interrupted, but it does so only infrequently—at critical moments while it's processing an interrupt or dispatching an exception, for example.

The kernel installs interrupt trap handlers to respond to device interrupts. Interrupt trap handlers transfer control either to an external routine (the ISR) that handles the interrupt or to an internal kernel routine that responds to the interrupt. Device drivers supply ISRs to service device interrupts, and the kernel provides interrupt handling routines for other types of interrupts.

In the following subsections, you'll find out how the hardware notifies the processor of device interrupts, the types of interrupts the kernel supports, the way device drivers interact with the kernel (as a part of interrupt processing), and the software interrupts the kernel recognizes (plus the kernel objects that are used to implement them).

Interrupt Types and Priorities)



Although interrupt controllers perform a level of interrupt prioritization, Windows imposes its own interrupt priority scheme known as *interrupt request levels* (IRQLs). The kernel represents IRQLs internally as a number from 0 through 31, with higher numbers representing higher-priority interrupts. Although the kernel defines the standard set of IRQLs for software interrupts, the HAL maps hardware-interrupt numbers to the IRQLs.

Interrupts are serviced in priority order, and a higher-priority interrupt preempts the servicing of a lower-priority interrupt. When a high-priority interrupt occurs, the processor saves the interrupted thread's state and invokes the trap dispatchers associated with the interrupt. The trap dispatcher raises the IRQL and calls the interrupt's service routine. After the service routine executes, the interrupt dispatcher lowers the processor's IRQL to where it was before the interrupt occurred and then loads the saved machine state. The interrupted thread resumes executing where it left off. When the kernel lowers the IRQL, lower-priority interrupts that were masked might materialize. If this happens, the kernel repeats the process to handle the new interrupts.

IRQL priority levels have a completely different meaning than thread-scheduling priorities. A scheduling priority is an attribute of a thread, whereas an IRQL is an attribute of an interrupt source, such as a keyboard or a mouse. In addition, each processor has an IRQL setting that changes as operating system code executes.

Each processor's IRQL setting determines which interrupts that processor can receive. IRQLs are also used to synchronize access to kernel-mode data structures. As a kernel-mode thread runs, it raises or lowers the processor's IRQL either directly by calling *KeRaiseIrql* and *KeLowerIrql* or, more commonly, indirectly via calls to functions that acquire kernel synchronization objects.

A kernel-mode thread raises and lowers the IRQL of the processor on which it's running, depending on what it's trying to do. For example, when an interrupt occurs, the trap handler (or perhaps the processor) raises the processor's IRQL to the assigned IRQL of the interrupt source. This elevation masks all interrupts at and below that IRQL (on that processor only), which ensures that the processor servicing the interrupt isn't waylaid by an interrupt at the same or a lower level. The masked interrupts are either handled by another processor or held back until the IRQL drops. Therefore, all components of the system, including the kernel and device drivers, attempt to keep the IRQL at passive level (sometimes called low level). They do this because device drivers can respond to

hardware interrupts in a timelier manner if the IRQL isn't kept unnecessarily elevated for long periods.

Because changing a processor's IRQL has such a significant effect on system operation, the change can be made only in kernel mode—user-mode threads can't change the processor's IRQL. This means that a processor's IRQL is always at passive level when it's executing user-mode code. Only when the processor is executing kernel-mode code can the IRQL be higher.

Each interrupt level has a specific purpose. For example, the kernel issues an *inter-processor interrupt* (IPI) to request that another processor perform an action, such as dispatching a particular thread for execution or updating its translation look-aside buffer cache. The system clock generates an interrupt at regular intervals, and the kernel responds by updating the clock and measuring thread execution time. If a hardware platform supports two clocks, the kernel adds another clock interrupt level to measure performance. The HAL provides a number of interrupt levels for use by interrupt-driven devices; the exact number varies with the processor and system configuration. The kernel uses software interrupts to initiate thread scheduling and to asynchronously break into a thread's execution.

Predefined IRQL's

Dispatchable or Preemptible?	IRQL	Value	Used For	Interruptible?
Neither	HIGH_LEVEL	31	Machine check, NMI	No
	POWER_LEVEL	30	Power failure	Partially
	IPI_LEVEL	29	Interprocessor Interrupt	
	CLOCK1_LEVEL	28	Clock Interrupt	
	PROFILE_LEVEL	27	Profiling Timer	Fully
		11-26	IRQs on HALx86	
Fully	DISPATCH_LEVEL	2	Dispatcher and DPCs	Fully
	APC_LEVEL	1	APCs and Page Fault Processing	
	PASSIVE_LEVEL	0	User/Kernel Threads	

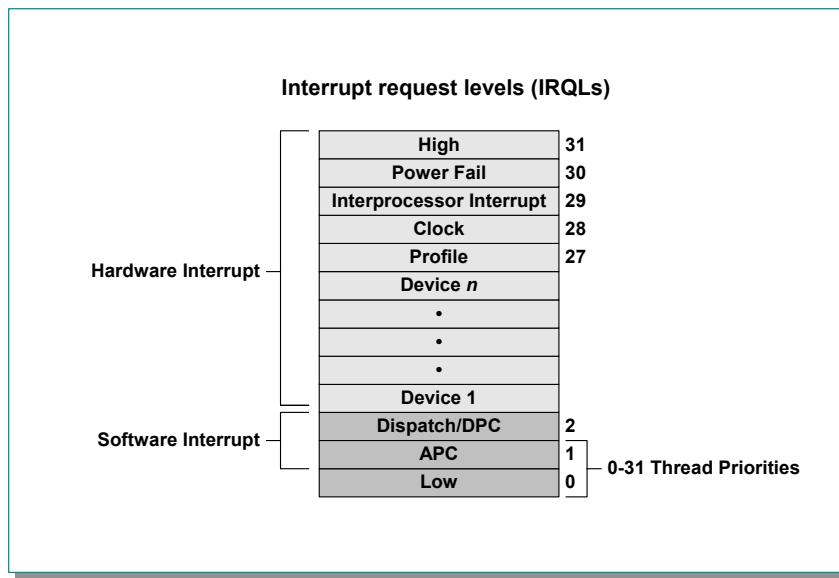
Let's take a closer look at the use of the predefined IRQLs, starting from the highest level:

- The kernel uses *high* level rarely, and only for very brief periods. It is used when halting the system via **KeBugCheckEx**, saving the processor state immediately after an interrupt, or holding a special type of serialization object called an "interlocked queue spinlock".
- Power fail* level originated in the original Microsoft Windows NT design documents, which specified the behavior of system power failure code, but this IRQL has never been used.
- Inter-processor interrupt* level is used to request another processor to perform an action, such as dispatching a particular thread for execution, updating the processor's translation look-aside buffer (TLB) cache, system shutdown, or system crash.
- Clock* level is used for the system's clock, which the kernel uses to track the time of day as well as to measure and allot CPU time to threads.
- The system's real-time clock uses *profile* level when kernel profiling, a performance measurement mechanism, is enabled. When kernel profiling is active, the kernel's profiling trap handler records the address of the code that was executing when the interrupt occurred. A table of address samples is constructed over time that tools can extract and analyze. The Windows resource kits include a tool called Kernel Profiler (Kernprof.exe) that you can use to configure and view profiling-generated statistics.
- The *device* IRQLs are used to prioritize device interrupts.
- DPC/dispatch-level* and *APC*-level interrupts are software interrupts that the kernel and device drivers generate. (APC level is actually a state of the thread, not the CPU.)

- The lowest IRQL, *passive* level, isn't really an interrupt level at all; it's the setting at which normal thread execution takes place. In this state, no interrupts are pending, and all interrupts are allowed to occur.

One important restriction on code running at DPC/dispatch level or above is that it can't wait on a dispatcher object if doing so would necessitate the scheduler to select another thread to execute. Another restriction is that only nonpaged memory can be accessed at IRQL DPC/dispatch level or higher. This rule is actually a side effect of the first restriction because attempting to access memory that isn't resident results in a page fault. When a page fault occurs, the memory manager initiates a disk I/O and then needs to wait for the file system driver to read the page in from disk. This wait would in turn require the scheduler to perform a context switch (perhaps to the idle thread if no user thread is waiting to run), thus violating the rule that the scheduler can't be invoked (because the IRQL is still DPC/dispatch level or higher at the time of the disk read). If either of these two restrictions is violated, the system crashes with an IRQL_NOT_LESS_OR_EQUAL crash code. Violating these restrictions is a common bug in device drivers.

Interrupt Levels vs. Thread Priority Levels

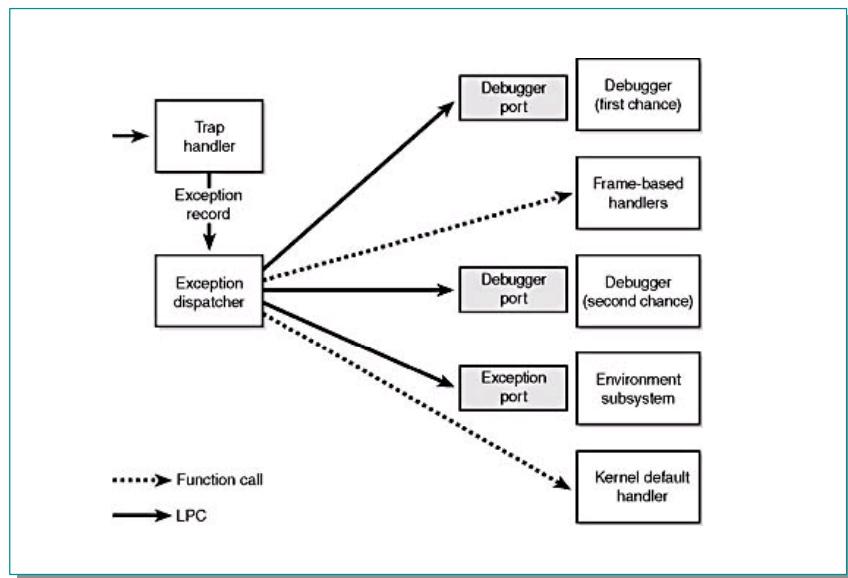


Interrupt Levels vs. Priority Levels

As illustrated above, all threads (whether they are running in user mode or kernel mode) run at IRQL 0 or 1. Threads normally run at IRQL 0. Only kernel mode APCs execute at IRQL 1, since they interrupt the execution of a thread. Because of this, no thread regardless of its priority, blocks hardware interrupts (although high-priority real-time threads can block the execution of important system threads).

Thread-scheduling decisions are made at IRQL 2 (called Dispatch level for thread dispatching). Thus, while the kernel is deciding which thread should run next, no thread can be running and possibly changing scheduling-related information (such as priorities). On a multiprocessor system, access to the thread-scheduling data structures is synchronized by acquiring the Dispatcher spinlock (*KiDispatcherLock*).

Exception Dispatching



In contrast to interrupts, which can occur at any time, exceptions are conditions that result directly from the execution of the program that is running. Win32 introduced a facility known as *structured exception handling*, which allows applications to gain control when exceptions occur. The application can then fix the condition and return to the place the exception occurred, unwind the stack (thus terminating execution of the subroutine that raised the exception), or declare back to the system that the exception isn't recognized and the system should continue searching for an exception handler that might process the exception.

On the x86, all exceptions have predefined interrupt numbers that directly correspond to the entry in the IDT that points to the trap handler for a particular exception.

All exceptions, except those simple enough to be resolved by the trap handler, are serviced by a kernel module called the *exception dispatcher*. The exception dispatcher's job is to find an exception handler that can "dispose of" the exception. Examples of architecture-independent exceptions that the kernel defines include memory access violations, integer divide-by-zero, integer overflow, floating-point exceptions, and debugger breakpoints. For a complete list of architecture-independent exceptions, consult the Win32 API reference documentation.

The kernel traps and handles some of these exceptions transparently to user programs. For example, encountering a breakpoint while executing a program being debugged generates an exception, which the kernel handles by calling the debugger. The kernel handles certain other exceptions by returning an unsuccessful status code to the caller.

When an exception occurs, whether it is explicitly raised by software or implicitly raised by hardware, a chain of events begins in the kernel. The CPU transfers control to the kernel trap handler, which creates a trap frame (as it does when an interrupt occurs). The trap frame allows the system to resume where it left off if the exception is resolved. The trap handler also creates an exception record that contains the reason for the exception and other pertinent information.

Debugger breakpoints are common sources of exceptions. Therefore, the first action the exception dispatcher takes is to see whether the process that incurred the exception has an associated debugger process. If so, it sends the first-chance debug message (via an LPC port) to the debugger port associated with the process that incurred the exception. (The

message is sent to the session manager process, which then dispatches it to the appropriate debugger process.)

If the process has no debugger process attached, or if the debugger doesn't handle the exception, the exception dispatcher switches into user mode and calls a routine to find a frame-based exception handler. If none is found, or if none handles the exception, the exception dispatcher switches back into kernel mode and calls the debugger again to allow the user to do more debugging. (This is called the second-chance notification.)

All Win32 threads have an exception handler declared at the top of the stack that processes unhandled exceptions. This exception handler is declared in the internal Win32 *start-of-process* or *start-of-thread* function. The start-of-process function runs when the first thread in a process begins execution. It calls the main entry point in the image. The start-of-thread function runs when a user creates additional threads. It calls the user-supplied thread start routine specified in the *CreateThread* call.

The generic code for these internal start functions is shown here:

```
void Win32StartOfProcess(
    LPTHREAD_START_ROUTINE lpStartAddr,
    LPVOID lpvThreadParm) {

    __try {

        DWORD dwThreadExitCode = lpStartAddr(lpvThreadParm);
        ExitThread(dwThreadExitCode);

    } __except(UnhandledExceptionFilter(
        GetExceptionInformation())) {

        ExitProcess(GetExceptionCode());
    }
}
```

Notice that the Win32 unhandled exception filter is called if the thread has an exception that it doesn't handle. This function looks in the registry in the *HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug* key to determine whether to run a debugger immediately or to ask the user first.

The default "debugger" on Windows is *\winnt\system32\DrWtsn32.exe* (Dr. Watson), which isn't really a debugger but rather a postmortem tool that captures the state of the application "crash" and records it in a log file (*Drwtsn32.log*) and a process crash dump file (*User.dmp*), both found by default in the *\Documents And Settings\All Users\Documents\DrWatson* folder.

The log file contains basic information such as the exception code, the name of the image that failed, a list of loaded DLLs, and a stack and instruction trace for the thread that incurred the exception.

The crash dump file contains the private pages in the process at the time of the exception. (The file doesn't include code pages from EXEs or DLLs.) This file can be opened by WinDbg, the Windows debugger that comes with the Debugging Tools for Windows package. Because the *User.dmp* file is overwritten each time a process crashes, unless you rename or copy the file after each process crash, you'll have only the latest one on your system.

If the debugger isn't running and no frame-based handlers are found, the kernel sends a message to the exception port associated with the thread's process. This exception port, if one exists, was registered by the environment subsystem that controls this thread. The exception port gives the environment subsystem, which presumably is listening at the port, the opportunity to translate the exception into an environment-specific signal or exception. Finally, if the kernel progresses this far in processing the exception and the subsystem doesn't handle the exception, the kernel executes a default exception handler that simply terminates the process whose thread caused the exception.

Debugger Commands: .trap, !pcr, !idt

- **.trap *trapFrameAddress***
 - Sets debugger context to that recorded in trap frame
 - Allows restoration of debugger register context to that at the time of a trap (exception or interrupt)
- **!pcr [*cpuNumber*]**
 - Displays Processor Control Region
 - Includes IRQL of the processor
- **!idt**
 - Displays interrupt dispatch table

The debugger includes a number of commands related to the concepts of trap and interrupt handling.

.trap *trapFrameAddress*

Sets the debugger's register context to that recorded in the indicated trap frame.

When a trap occurs, the system records the state of the CPU just prior to the trap in a data structure known as a *trap frame*. Eventually, if the trap is handled, the system returns to the interrupted code by restoring the CPU state from the trap frame.

If a failure occurs within the handling of the trap, the debugger's default register context will be that of the trap handling code in the operating system, not the code that raised or experienced the trap. The **.trap** command allows us to set the debugger's context to the state of the CPU at the time of the trap, allowing the reason for the trap to be analyzed.

For example, here is a register and stack trace from a crash dump file:

```
kd> r
eax=ffdfff13c ebx=00000000 ecx=f71efab8 edx=00000000 esi=f71efb38
edi=f71efae4
eip=80430059 esp=f71ef70c ebp=f71efac8 iopl=0          nv up ei ng nz na po
nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000
efl=00000286
nt!KiDispatchException+30e:
80430059 53          push     ebx
```

```

kd> kv
ChildEBP RetAddr  Args to Child
f71efac8 804624cb f71efae4 00000000 f71efb38 nt!KiDispatchException+0x30e
(FPO: [Non-Fpo])
f71efb30 8046247d 81288784 f71efb84 804b0b48
nt!CommonDispatchException+0x4d (FPO: [0,20,0])
f71efb30 804676c8 81288784 f71efb84 804b0b48
nt!KiUnexpectedInterruptTail+0x1f4 (FPO: [0,0] TrapFrame @ f71efb38)
f71efbd0 804672a2 00000001 00000000 8049b588 nt!ExFreePoolWithTag+0x342
(FPO: [Non-Fpo])
f71efbdc 8049b588 e1e40ea8 e1bbb80c 80495b70 nt!ExFreePool+0xb (FPO:
[1,0,0])
f71efbe8 80495b70 e1bbb810 e1bbb7f8 81437740 nt!SepTokenDeleteMethod+0x1b
(FPO: [1,0,1])
f71efc04 80444c3b3 e1bbb810 e1bbb810 81272020
nt!ObpRemoveObjectRoutine+0xd6 (FPO: [Non-Fpo])
f71efc28 80451ac4 00000002 81272020 f71efd3c nt!ObfDereferenceObject+0x149
(FPO: [Non-Fpo])
f71efc40 80451e5d 81272020 00000000 00000000 nt!PsImpersonateClient+0x191
(FPO: [Non-Fpo])
f71efc78 804983da 81272020 00000000 f71efd64
nt!PsAssignImpersonationToken+0x133 (FPO: [Non-Fpo])
f71efd4c 80461691 ffffffe 00000005 0054fa7c
nt!NtSetInformationThread+0x3a9 (FPO: [Non-Fpo])
f71efd4c 77f8c7c7 ffffffe 00000005 0054fa7c nt!KiSystemService+0xc4 (FPO:
[0,0] TrapFrame @ f71efd64)
0054fa64 010074c2 ffffffe 00000005 0054fa7c
ntdll!NtSetInformationThread+0xb (FPO: [4,0,0])
0054fa80 0100687b 0054fa9c 0054fac0 00160014
services!ScReleasePrivilege+0x18 (FPO: [Non-Fpo])
0054fa98 77d45178 00000015 02020202 00000001
services!RCloseServiceHandle+0x81 (FPO: [Non-Fpo])
0054fab0 77da1586 010067e0 0054fac4 00000001 RPCRT4!Invoke+0x30
0054fd10 77da1937 00000000 00000000 0054fdfc RPCRT4!NdrStubCall2+0x63d
(FPO: [Non-Fpo])
0054fd2c 77d453e2 0054fdfc 0008a288 0054fdfc RPCRT4!NdrServerCall2+0x17
(FPO: [Non-Fpo])
0054fd64 77d452ef 010065fd 0054fdfc 0054fe40 RPCRT4!DispatchToStubInC+0x84
(FPO: [Non-Fpo])
0054fdbc 77d45215 00000000 00000000 0054fe40
RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x100 (FPO: [Non-Fpo])

```

At first glance this stack trace might seem to imply that the problem occurred within the routine nt!KiDispatchException, because that is at the top of the stack, and the register display further indicates that the current instruction is within that routine. However, notice the “TrapFrame @” designation near the top of the stack (highlighted in bold). We supply the indicated address to the .trap command:

```

kd> .trap f71efb38
ErrCode = 00000002
eax=e1e40fc0 ebx=00000000 ecx=00000000 edx=00000000 esi=e1e40ea0
edi=81438428
eip=804676c8 esp=f71efbac ebp=f71efbd0 iopl=0 nv up ei pl zr na po
nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000
efl=00010246
nt!ExFreePoolWithTag+342:
804676c8 890a          mov     [edx],ecx
ds:0023:00000000=???????

```

A new register display appears, and a new “current instruction” designation as well. A subsequent stack trace shows the actual routine in which the trap occurred at the top of the stack:

```

kd> kv
*** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr  Args to Child
f71efbd0 804672a2 00000001 00000000 8049b588 nt!ExFreePoolWithTag+0x342
(FPO: [Non-Fpo])
f71efbdc 8049b588 e1e40ea8 e1bbb80c 80495b70 nt!ExFreePool+0xb (FPO:
[1,0,0])
f71efbe8 80495b70 e1bbb810 e1bbb7f8 81437740 nt!SepTokenDeleteMethod+0x1b
(FPO: [1,0,1])
f71efc04 8044c3b3 e1bbb810 e1bbb810 81272020
nt!ObpRemoveObjectRoutine+0xd6 (FPO: [Non-Fpo])
f71efc28 80451ac4 00000002 81272020 f71efd3c nt!ObfDereferenceObject+0x149
(FPO: [Non-Fpo])
f71efc40 80451e5d 81272020 00000000 00000000 nt!PsImpersonateClient+0x191
(FPO: [Non-Fpo])
f71efc78 804983da 81272020 00000000 f71efd64
nt!PsAssignImpersonationToken+0x133 (FPO: [Non-Fpo])
f71efd4c 80461691 ffffffe 00000005 0054fa7c
nt!NtSetInformationThread+0x3a9 (FPO: [Non-Fpo])
f71efd4c 77f8c7c7 ffffffe 00000005 0054fa7c nt!KiSystemService+0xc4 (FPO:
[0,0] TrapFrame @ f71efd64)
0054fa64 010074c2 ffffffe 00000005 0054fa7c
ntdll!NtSetInformationThread+0xb (FPO: [4,0,0])
0054fa80 0100687b 0054fa9c 0054fac0 00160014
services!ScReleasePrivilege+0x18 (FPO: [Non-Fpo])
0054fa98 77d45178 00000015 02020202 00000001
services!RCloseServiceHandle+0x81 (FPO: [Non-Fpo])
0054fab0 77da1586 010067e0 0054fac4 00000001 RPCRT4!Invoke+0x30
0054fd10 77da1937 00000000 00000000 0054fdfc RPCRT4!NdrStubCall12+0x63d
(FPO: [Non-Fpo])
0054fd2c 77d453e2 0054fdfc 0008a288 0054fdfc RPCRT4!NdrServerCall12+0x17
(FPO: [Non-Fpo])
0054fd64 77d452ef 010065fd 0054fdfc 0054fe40 RPCRT4!DispatchToStubInC+0x84
(FPO: [Non-Fpo])
0054fdbc 77d45215 00000000 00000000 0054fe40
RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x100 (FPO: [Non-Fpo])
0054fddc 77d4fa80 0054fdfc 00000000 0054fe40
RPCRT4!RPC_INTERFACE::DispatchToStub+0x5e (FPO: [Non-Fpo])
0054fe44 77d4f9d7 00000000 000fc328 001112f0
RPCRT4!OSF_SCALL::DispatchHelper+0xa4 (FPO: [Non-Fpo])
0054fe58 77d4f779 00000000 00000000 00000001
RPCRT4!OSF_SCALL::DispatchRPCCall+0x115 (FPO: [Non-Fpo])

```

!pcr [*cpuNumber*]

Displays the contents of the Processor Control Region for the indicated CPU.

Among the items displayed is the IRQL, Interrupt Request Level, at which the processor is executing.

!idt

Displays the interrupt dispatch table.

For every possible trap there is an entry in a table called the Interrupt Dispatch Table. When a trap occurs, the processor uses this entry to find the address of the correct trap handling routine. The **!idt** extension displays this table.

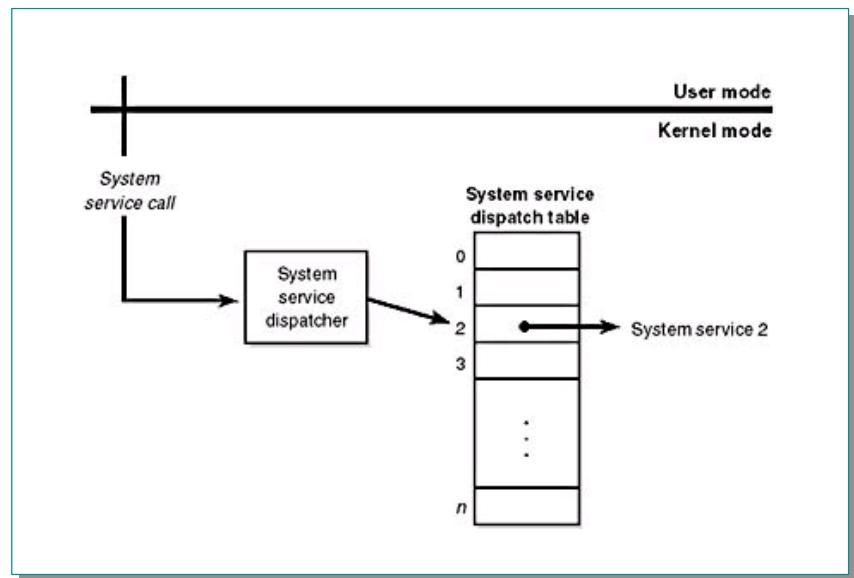
Note that for Windows 2000 target systems, you will first have to load a second kernel debugger extension library to make this command available:

```
kd> .load kdex2x86          needed for Windows 2000 only
kd> !idt

IDT for processor #0

00: 804625e6 (nt!KiTrap00)
01: 80462736 (nt!KiTrap01)
02: 0000144e
03: 80462a0e (nt!KiTrap03)
04: 80462b72 (nt!KiTrap04)
05: 80462cb6 (nt!KiTrap05)
06: 80462e1a (nt!KiTrap06)
07: 80463350 (nt!KiTrap07)
08: 000014a8
09: 8046370c (nt!KiTrap09)
0a: 80463814 (nt!KiTrap0A)
[...]
```

System Service Dispatching



The kernel's trap handlers dispatch interrupts, exceptions, and system service calls. In the preceding sections, you've seen how interrupt and exception handling work; in this section, you'll learn about system services.

In Windows 2000, a system service dispatch is triggered as a result of executing an *int 0x2E* instruction (46 decimal) on x86 processors. Because executing the *int* instruction results in a trap, Windows fills in entry 46 in the IDT to point to the system service dispatcher. The trap causes the executing thread to transition into kernel mode and enter the system service dispatcher. A numeric argument passed in the EAX processor register indicates the system service number being requested. The EBX register points to the list of parameters the caller passes to the system service.

The following code illustrates the generic code for a system service request:

```
NtWriteFile:
    mov  eax, 0x0E ; system service number for NtWriteFile
    mov  ebx, esp  ; point to parameters
    int  0x2E      ; execute system service trap
    ret  0x2C      ; pop parameters off stack and return
                    ; to caller
```

The system service dispatcher, *KiSystemService*, verifies the correct minimum number of arguments, copies the caller's arguments from the thread's user-mode stack to its kernel-mode stack (so that the user can't change the arguments as the kernel is accessing them), and then executes the system service.

System Calls in Newer Versions of Windows

If you are running Windows XP or a later version on Pentium III or higher, system calls are not made through *int 0x2E* calls, but rather via the SYSENTER and SYSEXIT instructions.

Memory Management

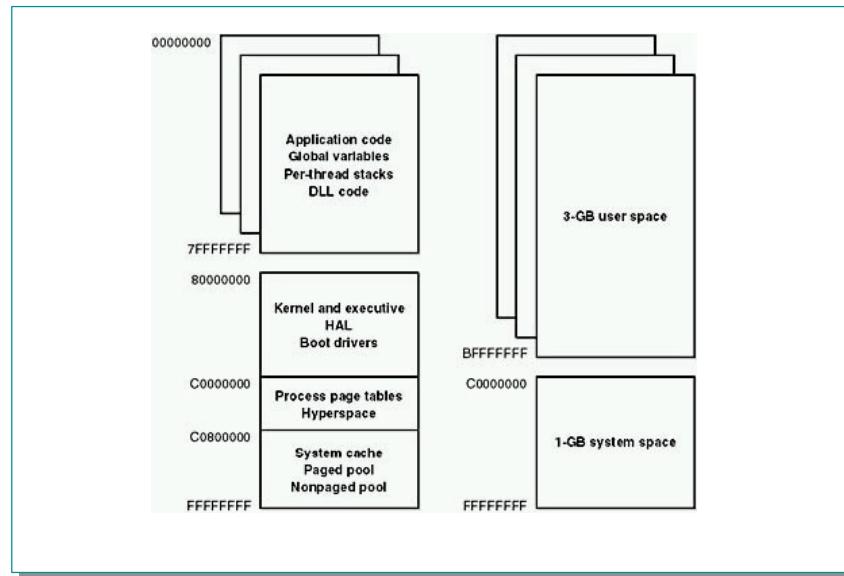
- **Services the Memory Manager Provides**
- **Address Space Layout**
- **System Address Space Layout**
- **Page Fault Handling**
- **System Memory Pools**

The memory manager provides a set of system services to allocate and free virtual memory, share memory between processes, map files into memory, flush virtual pages to disk, retrieve information about a range of virtual pages, change the protection of virtual pages, and lock the virtual pages into memory.

Like other Windows executive services, the memory management services allow their caller to supply a process handle, indicating the particular process whose virtual memory is to be manipulated. The caller can thus manipulate either its own memory or (with the proper permissions) the memory of another process. For example, if a process creates a child process, by default it has the right to manipulate the child process's virtual memory. Thereafter, the parent process can allocate, deallocate, read, and write memory on behalf of the child process by calling virtual memory services and passing a handle to the child process as an argument. This feature is used by subsystems to manage the memory of their client processes, and it is also key for implementing debuggers because debuggers must be able to read and write to the memory of the process being debugged.

The memory manager also provides a number of services, such as allocating and deallocating physical memory and locking pages in physical memory for direct memory access (DMA) transfers, to other kernel-mode components inside the executive as well as to device drivers. These functions begin with the prefix *Mm*. In addition, though not strictly part of the memory manager, the executive support routines that begin with *Ex* that are used to allocate and deallocate from the system heaps (paged and nonpaged pool) as well as to manipulate look-aside lists.

Address Space Layout

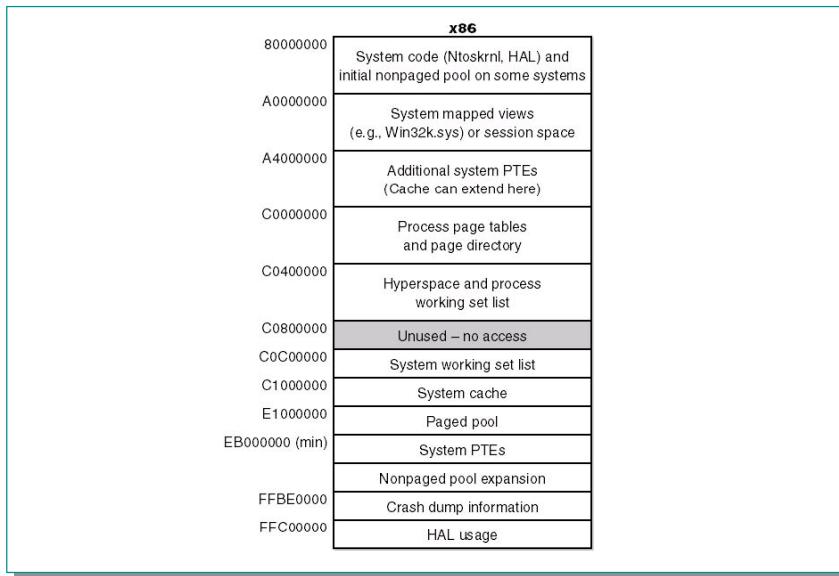


By default, each user process on the 32-bit version of Windows can have up to a 2 GB private address space; the operating system takes the remaining 2 GB. Windows Advanced Server and Windows Datacenter Server support a boot-time option that allows 3-GB user address spaces.

The 3-GB address space option (enabled by the /3GB flag in Boot.ini) gives processes a 3-GB address space (leaving 1 GB for system space). This feature was added as a short-term solution to accommodate the need for applications such as database servers to keep more data in memory than could be done with a 2-GB address space. The AWE functions provide a much better solution to the need for accessing more data than can fit in the limited 2-GB (or 3-GB) process address space.

For a process to access the full 3-GB address space, the image file must have the IMAGE_FILE_LARGE_ADDRESS_AWARE flag set in the image header. Otherwise, Windows reserves the third gigabyte so that the application won't see virtual addresses greater than 0x7FFFFFFF. You set this flag by specifying the linker flag /LARGEADDRESSAWARE when building the executable. This flag has no effect when running the application on a system with a 2-GB user address space. (If you boot Windows Professional or Windows Server with the /3GB switch, system space is reduced to 1 GB, but user processes are still limited to 2 GB, even if the large-address-space aware flag is set on an image that is run.)

System Address Space Layout



The x86 architecture has the following components in system space:

- **System code** Contains the operating system image, HAL, and device drivers used to boot the system.
- **System mapped views** Used to map Win32k.sys, the loadable kernel-mode part of the Win32 subsystem, as well as kernel-mode graphics drivers it uses
- **Session space** Used to map information specific to a user session. (Windows supports multiple user sessions when Terminal Services is installed.) The session working set list describes the parts of session space that are resident and in use.
- **Process page tables and page directory** Structures that describe the mapping of virtual addresses.
- **Hyperspace** A special region used to map the process working set list and to temporarily map other physical pages for such operations as zeroing a page on the free list (when the zero list is empty and a zero page is needed), invalidating page table entries in other page tables (such as when a page is removed from the standby list), and on process creation to set up a new process's address space.
- **System working set list** The working set list data structures that describe the system working set.
- **System cache** Virtual address space used to map files open in the system cache.
- **Paged pool** Pageable system memory heap.
- **System page table entries (PTEs)** Pool of system PTEs used to map system pages such as I/O space, kernel stacks, and memory descriptor lists. You can see how many system PTEs are available by examining the value of the Memory: Free System Page Table Entries counter in the Performance tool.

- **Nonpaged pool** Nonpageable system memory heap, usually existing in two parts—one in the lower end of system space and one in the upper end.
- **Crash dump information** Reserved to record information about the state of a system crash.
- **HAL usage** System memory reserved for HAL-specific structures.

Page Fault Handling

Reason for Fault	Result
Accessing a page that is not resident in memory but is on disk in a page file or mapped file	Allocate a physical page and read the desired page from disk and into the working set
Accessing a page that is on the standby or modified list	Transition the page to the process or system working set
Accessing a page that has no committed storage (reserved address space or address space that is not allocated)	Access violation
Accessing a page from user mode that can be accessed only in kernel mode	Access violation
Writing to a page that is read-only	Access violation
Accessing a demand-zero page	Add a zero filled page to the process working set
Writing to a guard page	Guard-page violation (if a reference to a user-mode stack, perform automatic stack expansion)
Writing to a copy-on-write page.	Make process-private copy of page and replace original in process or system working set
Referencing a page in system space that is valid but not in the process page directory	Copy page directory entry from master system page directory structure and dismiss exception
On a multiprocessor system, writing to a page that is valid but hasn't yet been written to	Set dirty bit in PTE

When the PTE valid bit is clear, this indicates that the desired page is for some reason not (currently) accessible to the process. This slide describes the types of invalid PTEs and how references to them are resolved.

A reference to an invalid page is called a *page fault*. The kernel trap handler dispatches this kind of fault to the memory manager fault handler (*MmAccessFault*) to resolve. This routine runs in the context of the thread that incurred the fault and is responsible for attempting to resolve the fault (if possible) or raise an appropriate exception. These faults can be caused by a variety of conditions.

System Memory Pools

- **Nonpaged pool** - Consists of ranges of system virtual addresses that are guaranteed to be resident in physical memory at all times and thus can be accessed at any time without incurring paging I/O.
- **Paged pool** - A region of virtual memory in system space that can be paged in and out of the system's working set. Because of this flexibility, there is no guarantee that an address within the paged portion of the system will not cause a page fault. For this reason, data structures that are accessed at interrupt request levels (IRQLs) of dispatch (DPC) level or above must be allocated from nonpaged pool.

At system initialization, the memory manager creates two types of dynamically sized memory pools that the kernel-mode components use to allocate system memory:

- **Nonpaged pool** Consists of ranges of system virtual addresses that are guaranteed to reside in physical memory at all times and thus can be accessed at any time (from any IRQL level and from any process context) without incurring a page fault. One of the reasons nonpaged pool is required is due to the rule that: page faults can't be satisfied at DPC/dispatch level or above.
- **Paged pool** A region of virtual memory in system space that can be paged in and out of the system. Device drivers that don't need to access the memory from DPC/dispatch level or above can use paged pool. It is accessible from any process context.

Both memory pools are located in the system part of the address space and are mapped in the virtual address space of every process. (In Table 7-10, you'll find out where in the system memory they start.) The executive provides routines to allocate and deallocate from these pools; for information on these routines, see the functions that start with *ExAllocatePool* in the Windows DDK documentation.

There are two types of nonpaged pools: one for general use and a small one (four pages) reserved for emergency use when nonpaged pool is full and the caller can't tolerate allocation failures. (This latter pool type should no longer be used; device drivers should be written to properly handle low memory conditions. Driver Verifier makes it easier to test such conditions.) Uniprocessor systems have three paged pools; multiprocessor systems have five. Having more than one paged pool reduces the frequency of system code blocking on simultaneous calls to pool routines. Both nonpaged and paged pool start at an initial size based on the amount of physical memory on the system and then grow, if necessary, up to a maximum size computed at system boot time. You can override the initial size of these pools by changing the values *NonPagedPoolSize* and *PagedPoolSize* in the registry key

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management from 0 (which causes the system to compute the size) to the size desired in bytes.

Debugger Commands: !vm, !memusage, !pte, !pfn, !dd, !pool

- **!vm** displays summary of virtual memory use
- **!memusage** displays summary of physical memory use
- **!pte address** displays address translation information (page directory and page table entries)
- **!pfn physicalPageNumber** displays usage information for a single physical page
- **!dd physAddress** displays memory via physical address
- **!pool address** displays boundaries of allocated and free pool regions “around” the address
- Includes “pool tag” for each region – see\Program Files\Debugging Tools for Windows\triage\pooltag.txt

!vm

Displays summary information about virtual memory use on the target system.

Example:

```
kd> !vm
*** Virtual Memory Usage ***
    Physical Memory:      4093  ( 32744 Kb)
    Available Pages:     2460  ( 19680 Kb)
    Modified Pages:       15   (   120 Kb)
    NonPagedPool Usage:  198   ( 1584 Kb)
    PagedPool 0 Usage:   387   ( 3096 Kb)
    PagedPool 1 Usage:   17   (   136 Kb)
    PagedPool 2 Usage:   15   (   120 Kb)
    PagedPool Usage:    419   ( 3352 Kb)
    Shared Commit:       3981  ( 31848 Kb)
    Shared Process:      41   (   328 Kb)
    Total Private:       31   (   248 Kb)
          smss.exe           25  (   200 Kb)
          System              6  (    48 Kb)
    PagedPool Commit:    419   ( 3352 Kb)
    Driver Commit:       62   (   496 Kb)
    Committed pages:    4662  ( 37296 Kb)
    Commit limit:       12619 (100952 Kb)
```

All memory use is listed in pages and in kilobytes. The most useful information in the **!vm** section for diagnosing problems is:

Parameter	Meaning
physical memory	Total physical memory in the system.
available pages	Number of pages of memory available on the system, both virtual and physical. If this number is low, it might indicate a problem with a process allocating too much virtual memory.
nonpaged pool usage	The amount of pages allocated to the nonpaged pool. The nonpaged pool is memory that cannot be swapped out to the paging file, so it must always occupy physical memory. This

number should rarely be larger than 10% of the total physical memory. If it is larger, this is usually an indication that there is a memory leak somewhere in the system.

!memusage

Displays summary information about physical memory use on the target system, and also displays information about open files.

Example:

```
kd> !memusage
loading PFN database
loading (99% complete)
    Zeroed:      36 (    144 kb)
    Free:        46 (    184 kb)
    Standby:    7821 ( 31284 kb)
    Modified:    642 (   2568 kb)
    ModifiedNoWrite: 0 (    0 kb)
    Active/Valid: 24206 ( 96824 kb)
    Transition:  0 (    0 kb)
    Unknown:     0 (    0 kb)
    TOTAL:     32751 (131004 kb)

Building kernel map
Finished building kernel map

Usage Summary (in Kb):
Control Valid Standby Dirty Shared Locked PageTables name
813caf08 296 156 0 0 0 mapped_file( hinv32.exe )
81125208 0 112 0 0 0 mapped_file( shell32.dll )
813ccb68 2764 0 0 0 0 No Name for File
8111b548 128 0 0 0 0 mapped_file( sms_def.mof )
812b6ac8 156 128 0 96 0 0 mapped_file( comctl32.dll )
810569a8 20 8 0 0 0 0 mapped_file( PIPCAP32.dll )
813c9108 2176 1920 0 0 0 0 No Name for File
```

!pte virtualAddress

Displays the page directory and page table entries for a given virtual address.

Example:

```
kd> !pte f71efbd0
F71EFBD0 - PDE at C0300F70          PTE at C03DC7BC
           contains 01000163          contains 070F0163
           pfn 1000 G-DA--KWW       pfn 70f0 G-DA--KWW
```

This shows that the physical page number, also known as the “page frame number” or PFN, that contains virtual addresses f71ef000 through f71effff, is PFN 70f0.

!dd physicalRange

Displays DWORDs in memory given their physical addresses. This takes the same options for *range* as the **dd** command, but the range is described via physical rather than virtual addresses.

For example:

```
kkd> dd f71efbd0
f71efbd0  f71efc04 804672a2 00000001 00000000
f71efbe0  8049b588 e1e40ea8 e1bbb80c 80495b70
f71efbf0  e1bbb810 e1bbb7f8 81437740 e1bbb810
f71efc00  00000000 f71efc28 8044c3b3 e1bbb810
f71efc10  e1bbb810 81272020 00000000 e1bbb7f8
f71efc20  00000000 001efd3c f71efc40 80451ac4
f71efc30  00000002 81272020 f71efd3c 001efd3c
f71efc40  f71efc78 80451e5d 81272020 00000000

kd> !dd 70f0bd0
# 70f0bd0 f71efc04 804672a2 00000001 00000000
# 70f0be0 8049b588 e1e40ea8 e1bbb80c 80495b70
# 70f0bf0 e1bbb810 e1bbb7f8 81437740 e1bbb810
# 70f0c00 00000000 f71efc28 8044c3b3 e1bbb810
```

```
# 70f0c10 e1bbb810 81272020 00000000 e1bbb7f8
# 70f0c20 00000000 001efd3c f71efc40 80451ac4
# 70f0c30 00000002 81272020 f71efd3c 001efd3c
# 70f0c40 f71efc78 80451e5d 81272020 00000000
```

We displayed the same area of memory, first given its virtual address and then (using the results from the previous `!pte` command) its physical address. The contents are the same.

!pool address

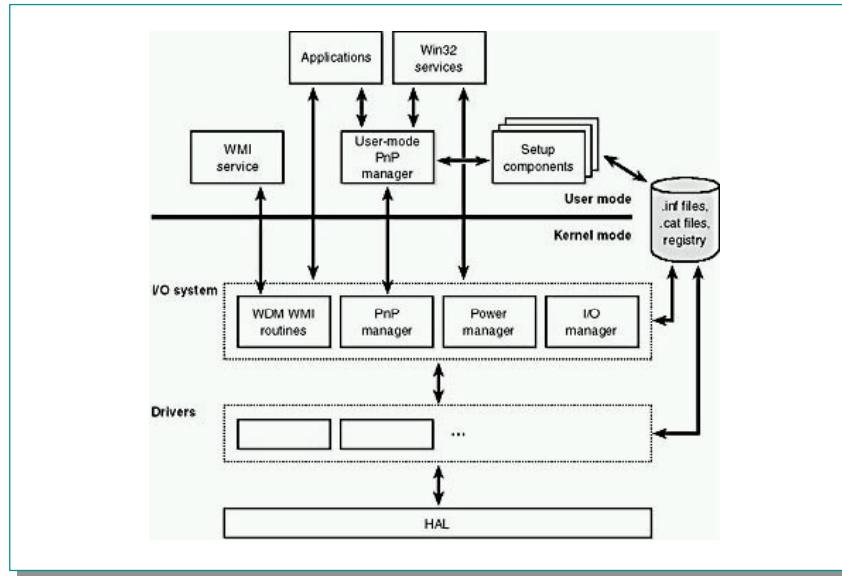
Displays information about pool structures before and, if possible, after the indicated address.

For example:

```
kd> !pool e1e40ea8
e1e40000 size: 80 previous size: 0 (Free) CMDa
e1e40080 size: 60 previous size: 80 (Allocated) CMkb (Protected)
e1e400e0 size: 20 previous size: 60 (Free) CMNb
e1e40100 size: a0 previous size: 20 (Allocated) Ntfo
e1e401a0 size: 20 previous size: a0 (Free) SeSd
e1e401c0 size: 60 previous size: 20 (Allocated) CMkb (Protected)
e1e40220 size: 60 previous size: 60 (Allocated) Sect (Protected)
e1e40280 size: 60 previous size: 60 (Allocated) CMkb (Protected)
e1e402e0 size: 80 previous size: 60 (Allocated) Gla@
e1e40360 size: a0 previous size: 80 (Free) CMkb
e1e40400 size: 20 previous size: a0 (Allocated) Ntf0
e1e40420 size: 20 previous size: 20 (Allocated) NtFa
e1e40440 size: 80 previous size: 20 (Allocated) Gla@
e1e404c0 size: 20 previous size: 80 (Free) CMNb
e1e404e0 size: 20 previous size: 20 (Allocated) NtFa
e1e40500 size: 20 previous size: 20 (Free) Toke
e1e40520 size: 40 previous size: 20 (Allocated) Process: 810abae0
e1e40560 size: 2a0 previous size: 40 (Allocated) Gla:
e1e40800 size: 460 previous size: 2a0 (Allocated) Gh 8
e1e40c60 size: 60 previous size: 460 (Allocated) CMkb (Protected)
e1e40cc0 size: 60 previous size: 60 (Allocated) ObNm
e1e40d20 size: c0 previous size: 60 (Free) MmSt
e1e40de0 size: a0 previous size: c0 (Allocated) MmSt
e1e40e80 size: 20 previous size: a0 (Free) CMkb
*e1e40ea0 size: 120 previous size: 20 (Lookaside) *SeTd
Bad allocation size @e1e40fc0, zero is invalid
```

This seems to indicate that the pool region beginning at address `e1e40fc0` is corrupted.

I/O System Components

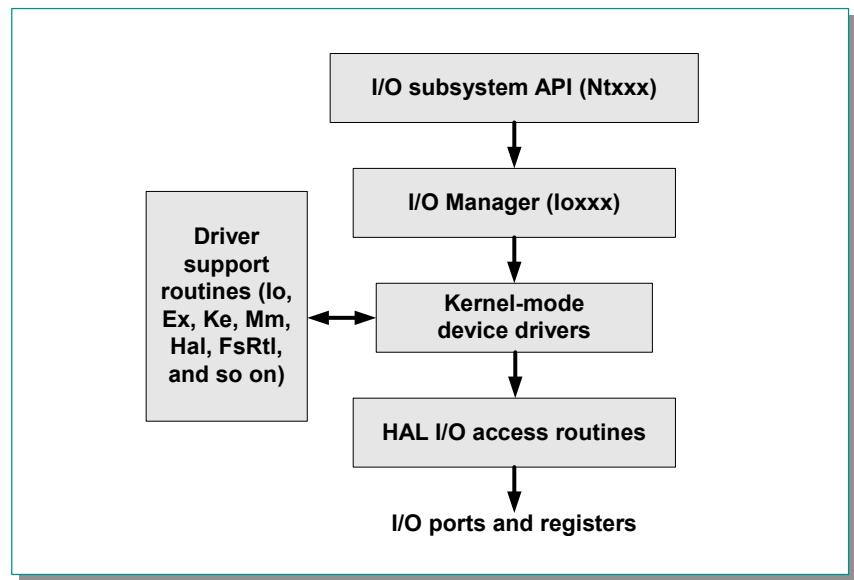


The Windows I/O system consists of several executive components as well as device drivers, which are shown above.

- The I/O manager connects applications and system components to virtual, logical, and physical devices, and defines the infrastructure that supports device drivers.
- A device driver typically provides an I/O interface for a particular type of device. Device drivers receive commands routed to them by the I/O manager that are directed at devices they manage, and they inform the I/O manager when those commands complete. Device drivers often use the I/O manager to forward I/O commands to other device drivers that share in the implementation of a device's interface or control.
- The PnP manager works closely with the I/O manager and a type of device driver called a *bus driver* to guide the allocation of hardware resources as well as to detect and respond to the arrival and removal of hardware devices. The PnP manager and bus drivers are responsible for loading a device's driver when the device is detected. When a device is added to a system that doesn't have an appropriate device driver, the executive Plug and Play component calls on the device installation services of a user-mode PnP manager.
- The power manager also works closely with the I/O manager to guide the system, as well as individual device drivers, through power-state transitions.
- Windows Management Instrumentation (WMI) support routines, called the Windows Driver Model (WDM) WMI provider, allow device drivers to indirectly act as providers, using the WDM WMI provider as an intermediary to communicate with the WMI service in user mode.
- The registry serves as a database that stores a description of basic hardware devices attached to the system as well as driver initialization and configuration settings.

- INF files, which are designated by the .inf extension, are driver installation files. INF files are the link between a particular hardware device and the driver that assumes primary control of the device. They are made up of script like instructions describing the device they correspond to, the source and target locations of driver files, required driver-installation registry modifications, and driver dependency information. Digital signatures that Windows uses to verify that a driver file has passed testing by the Microsoft Windows Hardware Quality Lab (WHQL) are stored in .cat files.
- The hardware abstraction layer (HAL) insulates drivers from the specifics of the processor and interrupt controller by providing APIs that hide differences between platforms. In essence, the HAL is the bus driver for all the devices on the computer's motherboard that aren't controlled by other drivers.

I/O System Structure and Model



In Windows, threads perform I/O on virtual files. The operating system abstracts all I/O requests as operations on a virtual file, hiding the fact that the target of an I/O operation might not be a file-structured device. This abstraction generalizes an application's interface to devices. A virtual file refers to any source or destination for I/O that is treated as if it were a file (such as files, directories, pipes, and mailslots). All data that is read or written is regarded as a simple stream of bytes directed to these virtual files. User-mode applications (whether Win32, POSIX, or OS/2) call documented functions, which in turn call internal I/O system functions to read from a file, write to a file, and perform other operations. The I/O manager dynamically directs these virtual file requests to the appropriate device driver. This slide illustrates the basic structure of a typical I/O request flow.

The I/O Manager

The *I/O manager* defines the orderly framework, or model, within which I/O requests are delivered to device drivers. The I/O system is *packet driven*. Most I/O requests are represented by an *I/O request packet* (IRP), which travels from one I/O system component to another. (Fast I/O is the exception; it doesn't use IRPs.) The design allows an individual application thread to manage multiple I/O requests concurrently. An IRP is a data structure that contains information completely describing an I/O request.

The I/O manager creates an IRP that represents an I/O operation, passing a pointer to the IRP to the correct driver and disposing of the packet when the I/O operation is complete. In contrast, a driver receives an IRP, performs the operation the IRP specifies, and passes the IRP back to the I/O manager, either for completion or to be passed on to another driver for further processing.

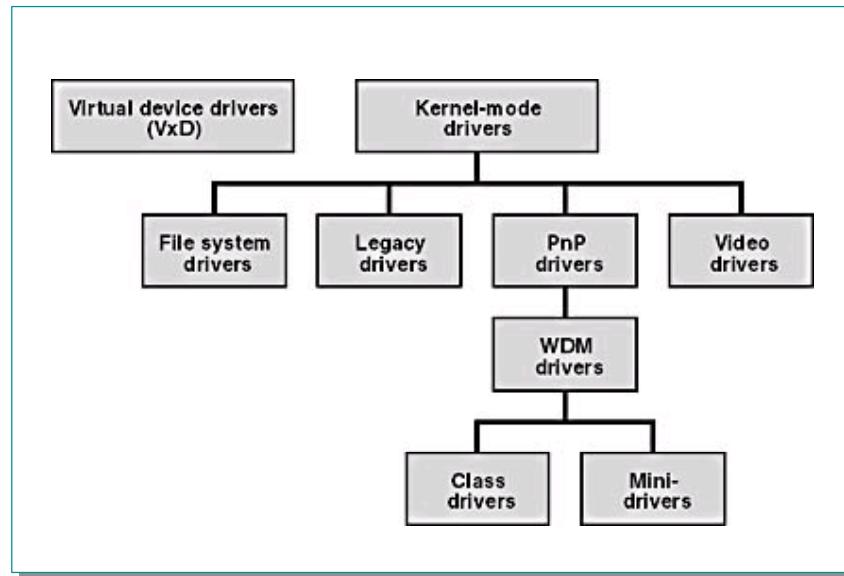
In addition to creating and disposing of IRPs, the I/O manager supplies code that is common to different drivers and that the drivers call to carry out their I/O processing. By consolidating common tasks in the I/O manager, individual drivers become simpler and more compact. For example, the I/O manager provides a function that allows one driver to call other drivers. It also manages buffers for I/O requests, provides timeout support for drivers, and records which installable file systems are loaded into the operating system. There are close to a hundred different routines in the I/O manager that can be called by device drivers.

The I/O manager also provides flexible I/O services that allow environment subsystems, such as Win32 and POSIX, to implement their respective I/O functions. These services include sophisticated services for asynchronous I/O that allow developers to build scalable high-performance server applications.

The uniform, modular interface that drivers present allows the I/O manager to call any driver without requiring any special knowledge of its structure or internal details. As we stated earlier, the operating system treats all I/O requests as if they were directed at a file; the driver converts the requests from requests made to a virtual file to hardware-specific requests. Drivers can also call each other (using the I/O manager) to achieve layered, independent processing of an I/O request.

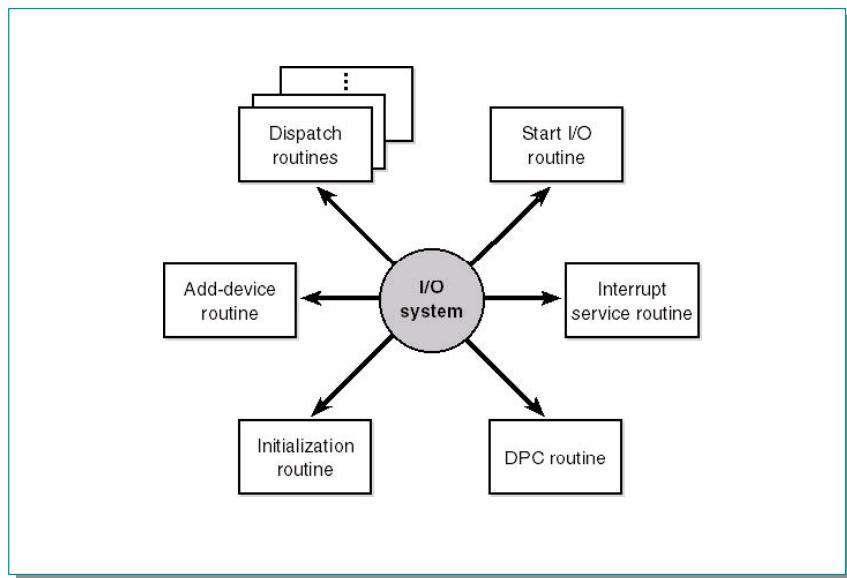
Besides the normal open, close, read, and write functions, the Windows I/O system provides several advanced features, such as asynchronous, direct, buffered, and scatter/gather I/O.

Device Drivers



- A *virtual device driver* (VDD) is a user-mode component that allows Microsoft® MS-DOS®-based applications to access hardware on x86 platforms. A VDD relies on the I/O permission mask to trap port access, and it essentially simulates the operation of hardware for the benefit of applications that were originally programmed to talk directly to hardware on a bare machine. Although this kind of driver shares a name and a purpose with a kind of driver used in Windows 98, it's a different animal altogether. We use the acronym VDD for this kind of driver and the acronym VxD for the Windows 98 driver to distinguish the two.
- The category of *kernel-mode drivers* includes many subcategories. A *PnP driver* is a kernel-mode driver that understands the Plug and Play (PnP) protocols of Windows.
- A *WDM driver* is a PnP driver that also understands power management protocols and is source-compatible with both Windows 98/Me and Windows 2000 and later operating systems. Within the category of WDM drivers, you can also distinguish between *class drivers*, which manage a device belonging to some well-defined class of device, and *miniport drivers*, which supply vendor-specific help to a class driver.
- *Video drivers* are kernel-mode drivers for display. In Windows 2000 and later, printer drivers are user-mode drivers. Video drivers are used for devices whose primary characteristic is that they render visual data.
- *File system drivers* implement the standard PC file system model (which includes the concept of a hierarchical directory structure containing named files) on local hard disks or over network connections.
- *Legacy device drivers* are kernel-mode drivers that directly control a hardware device without help from other drivers. This category essentially includes drivers for earlier versions of Windows NT that are running without change in Windows 2000 or later operating systems.

Structure of a Driver



The I/O system drives the execution of device drivers. Device drivers consist of a set of routines that are called to process the various stages of an I/O request.

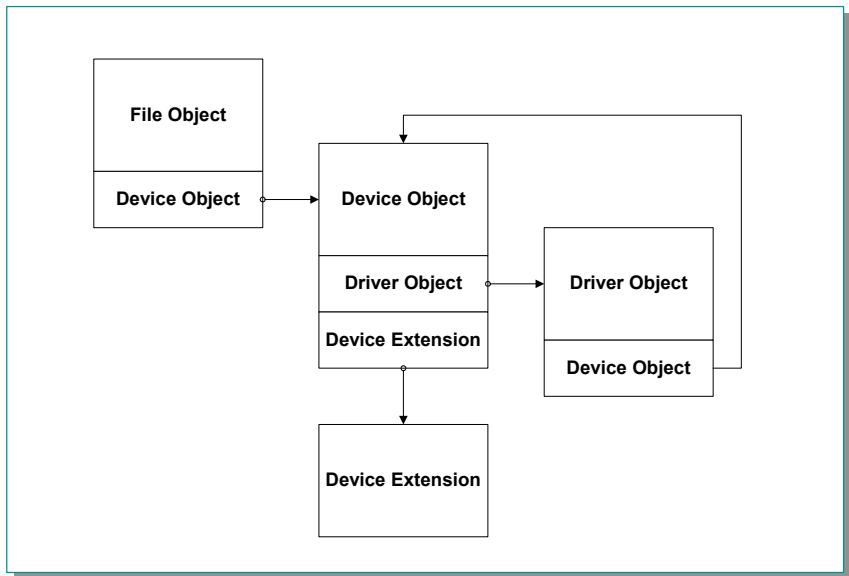
- **An initialization routine** The I/O manager executes a driver's initialization routine, which is typically named *DriverEntry*, when it loads the driver into the operating system. The routine fills in system data structures to register the rest of the driver's routines with the I/O manager and performs any global driver initialization that's necessary.
- **An add-device routine** A driver that supports Plug and Play implements an add-device routine. The PnP manager sends a driver notification via this routine whenever a device for which the driver is responsible is detected. In this routine, a driver typically allocates a device object to represent the device.
- **A set of dispatch routines** Dispatch routines are the main functions that a device driver provides. Some examples are open, close, read, and write and any other capabilities the device, file system, or network supports. When called on to perform an I/O operation, the I/O manager generates an IRP and calls a driver through one of the driver's dispatch routines.
- **A start I/O routine** The driver can use a start I/O routine to initiate a data transfer to or from a device. This routine is defined only in drivers that rely on the I/O manager for IRP serialization. The I/O manager serializes IRPs for a driver by ensuring that the driver processes only one IRP at a time. Some drivers process multiple IRPs concurrently, but serialization makes sense for many drivers, such as a keyboard driver.
- **An interrupt service routine (ISR)** When a device interrupts, the kernel's interrupt dispatcher transfers control to this routine. In the Windows I/O model, ISRs run at device interrupt request level (DIRQL), so they perform as little work as possible to avoid blocking lower-level interrupts unnecessarily. An ISR queues a deferred procedure call (DPC), which runs at a lower IRQL (DPC/dispatch level), to execute the remainder of interrupt processing. (Only drivers for interrupt-driven devices have ISRs; a file system driver, for example, doesn't have one.)

- **An interrupt-servicing DPC routine** A DPC routine performs most of the work involved in handling a device interrupt after the ISR executes. The DPC routine executes at a lower IRQL DPC/dispatch level than that of the ISR, which runs at device level, to avoid blocking other interrupts unnecessarily. A DPC routine initiates I/O completion and starts the next queued I/O operation on a device.

Although the following routines aren't shown above, they're found in many types of device drivers:

- **One or more I/O completion routines** A layered driver might have I/O completion routines that will notify it when a lower-level driver finishes processing an IRP. For example, the I/O manager calls a file system driver's I/O completion routine after a device driver finishes transferring data to or from a file. The completion routine notifies the file system driver about the operation's success, failure, or cancellation, and it allows the file system driver to perform cleanup operations.
- **A cancel I/O routine** If an I/O operation can be canceled, a driver can define one or more cancel I/O routines. When the driver receives an IRP for an I/O request that can be canceled, it assigns a cancel routine to the IRP. If a thread that issues an I/O request exits before the request is completed or cancels the operation (with the *CancelIo* Win32 function, for example), the I/O manager executes the IRP's cancel routine if one is assigned to it. A cancel routine is responsible for performing whatever steps are necessary to release any resources acquired during the processing that has already taken place for the IRP as well as completing the IRP with a canceled status.
- **An unload routine** An unload routine releases any system resources a driver is using so that the I/O manager can remove them from memory. Any resources acquired in the initialization routine are usually released in the unload routine. A driver can be loaded and unloaded while the system is running.
- **A system shutdown notification routine** This routine allows driver cleanup on system shutdown.
- **Error-logging routines** When unexpected errors occur (for example, when a disk block goes bad), a driver's error-logging routines note the occurrence and notify the I/O manager. The I/O manager writes this information to an error log file.

I/O Data Structures



Data Structures

Four primary data structures represent I/O requests: file objects, driver objects, device objects, and I/O request packets (IRPs).

Each of these structures is defined in the DDK header file `\DDK\INC\NTDDK.H` as well as in the DDK documentation. You can display them with a variety of debugger commands, such as `!devobj`, `!drvobj`, and `!irp`.

File Object

A user-mode-visible object that represents an open instance of a file, device, directory, volume, etc. A file object is accessed through the file handle returned by a protected subsystem's function, which calls down to a system service that opens (or creates) a file object. Callers of the I/O system services can wait on a file handle. At any given moment, several file objects can be associated with a single, shared data file, but each such file object has a unique handle and an object-specific value for the current file pointer.

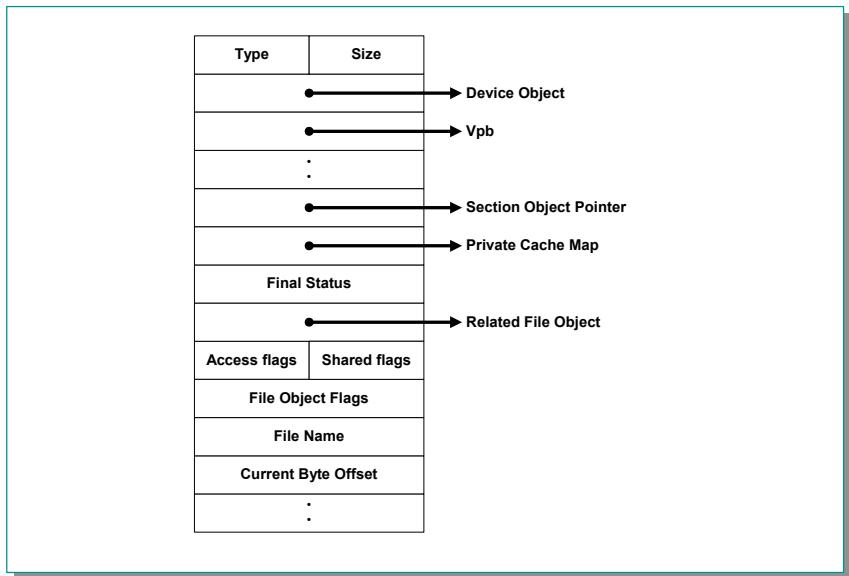
Driver Object

A kernel-mode-only object representing a driver's load image, used by the I/O Manager to locate certain entry points in the driver.

Device Object

A kernel-mode object, defined by the I/O Manager, that represents a physical, logical, or virtual device. Each driver calls `IoCreateDevice` to create and initialize a device object for each physical, logical, or virtual device that the driver services. PnP drivers create three kinds of device objects: bus drivers create PDOs, function drivers create FDOs, and filter drivers create FiDOs. A device is "visible" to end users as a named file object, and to user-mode code (protected subsystems) through a named device interface.

File Object



Files clearly fit the criteria for objects in Windows: they are system resources that threads in two or more user-mode processes can share; they can have names; they are protected by object-based security; and they support synchronization. Although most shared resources in Windows are memory-based resources, most of those that the I/O system manages are located on physical devices or are actual physical devices. Despite this difference, shared resources in the I/O system, like those in other components of the Windows executive, are manipulated as objects.

File objects provide a memory-based representation of shareable physical resources (except named pipes and mailslots, which are memory-based rather than physical resources). File objects also represent these resources in the Windows I/O system. This slide lists some of the file object's attributes. For specific field declarations and sizes, see the structure definition for FILE_OBJECT in NTDDK.H.

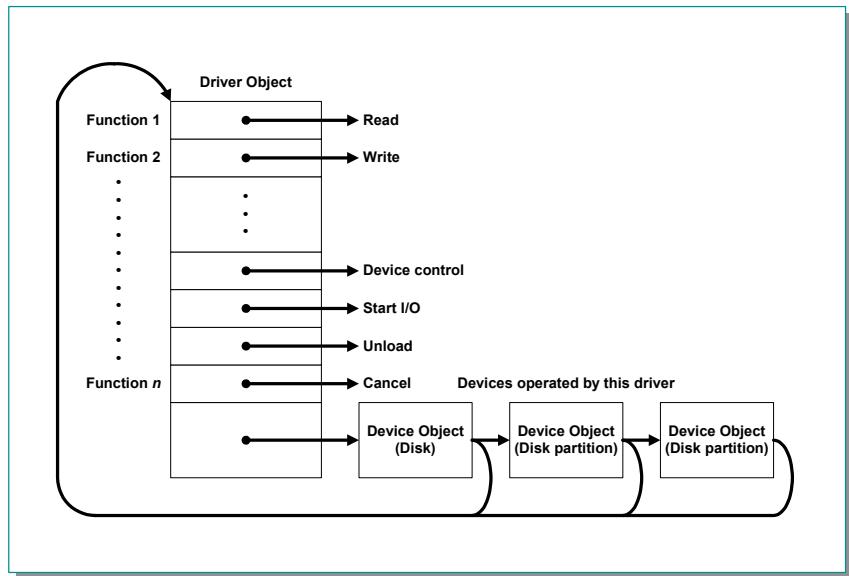
Like other executive objects, file objects are protected by a security descriptor that contains an access control list (ACL). The I/O manager consults the security subsystem to determine whether the file's ACL allows the process to access the file in the way its thread is requesting. If it does, the object manager grants the access and associates the granted access rights with the file handle that it returns. If this thread or another thread in the process needs to perform additional operations not specified in the original request, the thread must open another handle, which prompts another security check.

Because a file object is a memory-based representation of a shareable resource and not the resource itself, it is different from other executive objects. A file object contains only data that is unique to an object handle, whereas the file itself contains the data or text to be shared. Each time a thread opens a file handle, a new file object is created with a new set of handle-specific attributes. For example, the attribute byte offset refers to the location in the file at which the next read or write operation using that handle will occur. Each thread that opens a handle to a file has a private byte offset even though the underlying file is shared. A file object is also unique to a process, except when a process duplicates a file handle to another process or when a child process inherits a file handle from a parent process. In these situations, the two processes have separate handles that refer to the same file object.

Although a file handle might be unique to a process, the underlying physical resource is not. Therefore, as when using any shared resource, threads must synchronize their access to shareable files, file directories, or devices. If a thread is writing to a file, for

example, it should specify exclusive write access when opening the file handle to prevent other threads from writing to the file at the same time. Alternatively, by using the Win32 *LockFile* function, it could lock portions of the file while writing to it.

Driver Objects and Device Objects



When a thread opens a handle to a file object, the I/O manager must determine from the file object's name which driver (or drivers) it should call to process the request. Furthermore, the I/O manager must be able to locate this information the next time a thread uses the same file handle. The following system objects fill this need:

- A *driver object* represents an individual driver in the system and records for the I/O manager the address of each of the driver's dispatch routines (entry points).
- A *device object* represents a physical, logical, or virtual device on the system and describes its characteristics, such as the alignment it requires for buffers and the location of its device queue to hold incoming I/O request packets.

The I/O manager creates a driver object when a driver is loaded into the system, and it then calls the driver's initialization routine, which fills in the object with the driver's entry points. One of these is the driver's AddDevice routine. This routine is called later during device enumeration, once for each device discovered that is to be operated by this driver. The AddDevice routine creates a device object to represent the newly-discovered device. The device objects are attached to the driver object via a linked list, as shown above.

When a file is opened, the filename includes the name of the device object on which the file resides. For example, the name \Device\Floppy0\myfile.dat refers to the file *myfile.dat* on the floppy disk drive A. The substring \Device\Floppy0 is the name of the internal Windows device object representing that floppy disk drive. When opening *myfile.dat*, the I/O manager creates a file object and stores a pointer to the Floppy0 device object in the file object and then returns a file handle to the caller. Thereafter, when the caller uses the file handle, the I/O manager can find the Floppy0 device object directly. Keep in mind that internal Windows device names can't be used in Win32 application –instead, the device name must appear in a special directory in the object manager's namespace, \?? (Formerly named \DosDevices). This directory contains symbolic links to the real, internal Windows device names. Device drivers are responsible for creating links in this directory so that their devices will be accessible to Win32 applications. You can examine or even change these links programmatically with the Win32 *QueryDosDevice* and *DefineDosDevice* functions.

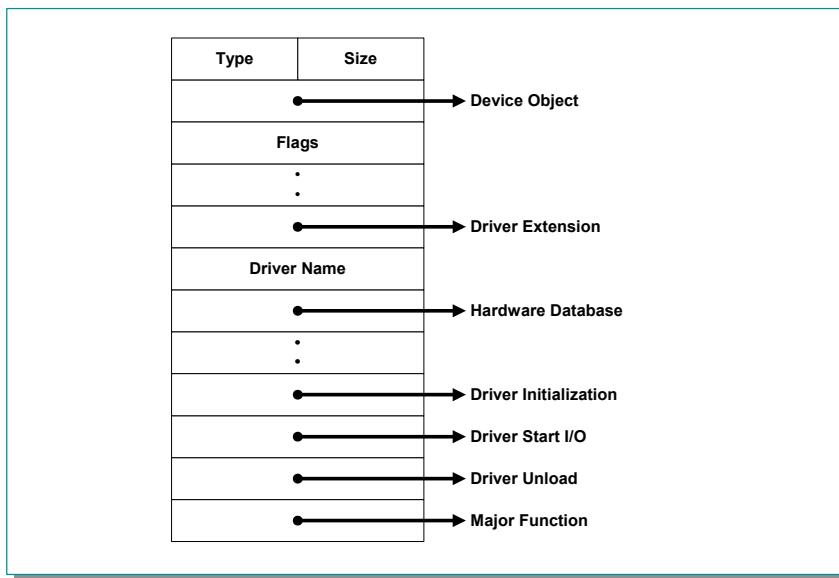
A device object points back to its driver object, which is how the I/O manager knows which driver routine to call when it receives an I/O request. It uses the device object to

find the driver object representing the driver that services the device. It then indexes into the driver object using the function code supplied in the original request; each function code corresponds to a driver entry point.

A driver object often has multiple device objects associated with it. The list of device objects represents the physical, logical, and virtual devices that the driver controls. For example, each partition of a hard disk has a separate device object that contains partition-specific information. However, the same hard disk driver is used to access all partitions. When a driver is unloaded from the system, the I/O manager uses the queue of device objects to determine which devices will be affected by the removal of the driver.

Using objects to record information about drivers prevents the I/O manager from needing to know details about individual drivers. The I/O manager merely follows a pointer to locate a driver, which provides a layer of portability and allows new drivers to be loaded easily. Representing devices and drivers with different objects also makes it easy for the I/O system to assign drivers to control additional or different devices if the system configuration changes.

Driver Object



The **DRIVER_OBJECT** Structure

Each driver object represents the image of a loaded kernel-mode driver. A pointer to the driver object is an input parameter to a driver's **DriverEntry**, **AddDevice**, and optional **Reinitialize** routines and to its **Unload** routine, if any.

Key fields in the Driver Object

DeviceObject

Points to the device object(s) created by the driver. This field is automatically updated when the driver calls **IoCreateDevice** successfully. A driver can this field and the **NextDevice** field of the **DEVICE_OBJECT** to step through the list of all the device objects the driver created.

HardwareDatabase

Points to the **\Registry\Machine\Hardware** path to the hardware configuration information in the registry.

DriverInit

Is the entry point for the **DriverEntry** routine, which is set up by the I/O Manager.

DriverStartIo

Is the entry point for the driver's **StartIo** routine, if any, which is set by the **DriverEntry** routine when the driver initializes. If a driver has no **StartIo** routine, this field is NULL.

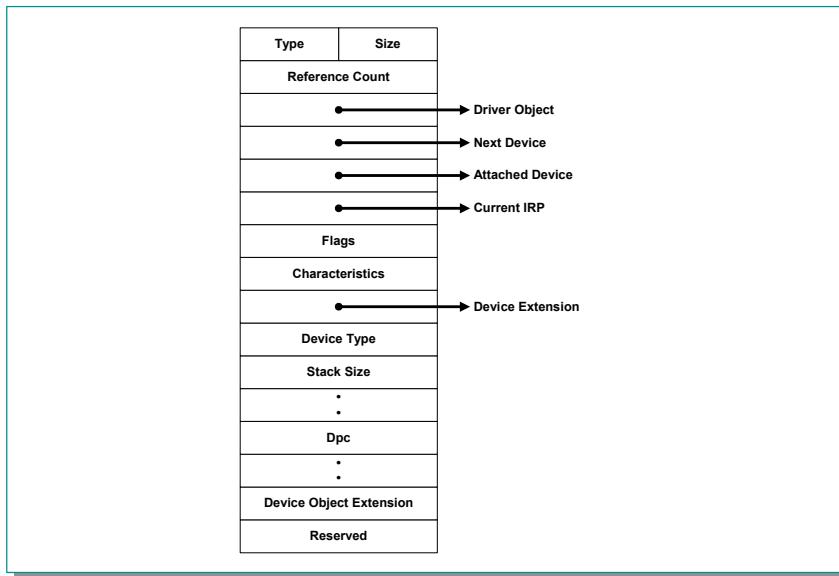
DriverUnload

Is the entry point for the driver's **Unload** routine, if any, which is set by the **DriverEntry** routine when the driver initializes. If a driver has no **Unload** routine, this field is NULL.

MajorFunction

Is an array of one or more entry points for the driver's Dispatch routines. Each driver must set at least one Dispatch entry point in this array for the **IRP_MJ_XXX** requests that the driver handles. A driver can set as many separate Dispatch entry points as the **IRP_MJ_XXX** codes that the driver handles.

Device Object



The **DEVICE_OBJECT** Structure

A device object represents a logical, virtual, or physical device for which a loaded driver handles I/O requests. Every kernel-mode driver must call **IoCreateDevice** one or more times from its AddDevice routine to create its device object(s).

Key Fields in the Device Object

DriverObject

Points to the driver object, representing the driver's loaded image.

NextDevice

Points to the next device object, if any, created by the same driver.

CurrentIrp

Points to the current IRP if the driver has a StartIo routine whose entry point was set in the driver object and if the driver is currently processing IRP(s). Otherwise, this field is NULL.

Flags

Device drivers OR this field in their newly created device objects with either of the following values: DO_BUFFERED_IO or DO_DIRECT_IO.

Characteristics

Set when a driver calls **IoCreateDevice** with one of the following values, as appropriate: FILE_REMOVABLE_MEDIA, FILE_READ_ONLY_DEVICE, FILE_FLOPPY_DISKETTE, FILE_WRITE_ONCE_MEDIA, FILE_DEVICE_SECURE_OPEN.

DeviceExtension

Points to the device extension. The structure and contents of the device extension are driver-defined. The size is driver-determined, specified in the driver's call to

IoCreateDevice

Most driver routines that process IRPs are given a pointer to the device object so the device extension is usually every driver's primary global storage area and frequently a driver's only global storage area for objects, resources, and any state the driver maintains about the I/O requests it handles.

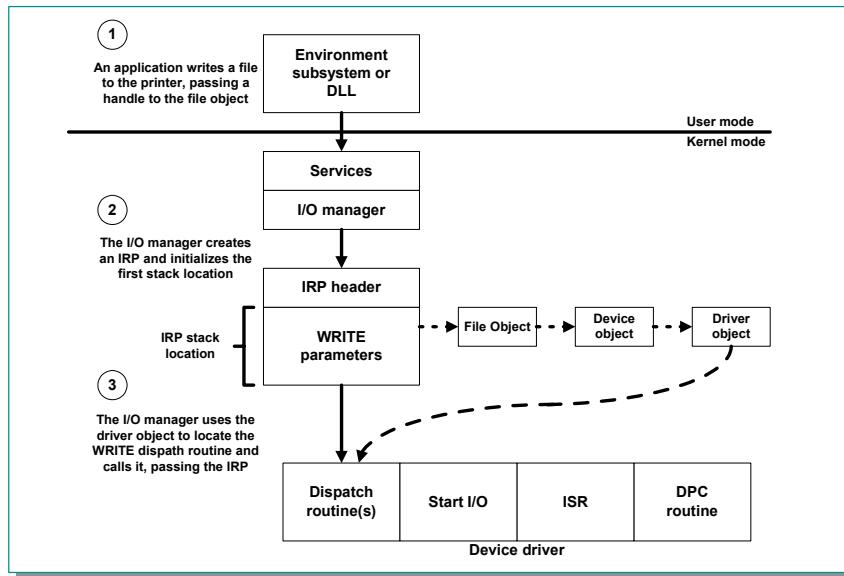
DeviceType

Set when a driver calls **IoCreateDevice** as appropriate for the type of underlying device. A driver writer can define a new FILE_DEVICE_XXX with a value in the customer range 32768 to 65535 if none of the system-defined values describes the type of the new device.

StackSize

Specifies the minimum number of stack locations in IRPs to be sent to this driver.

Processing an I/O Request



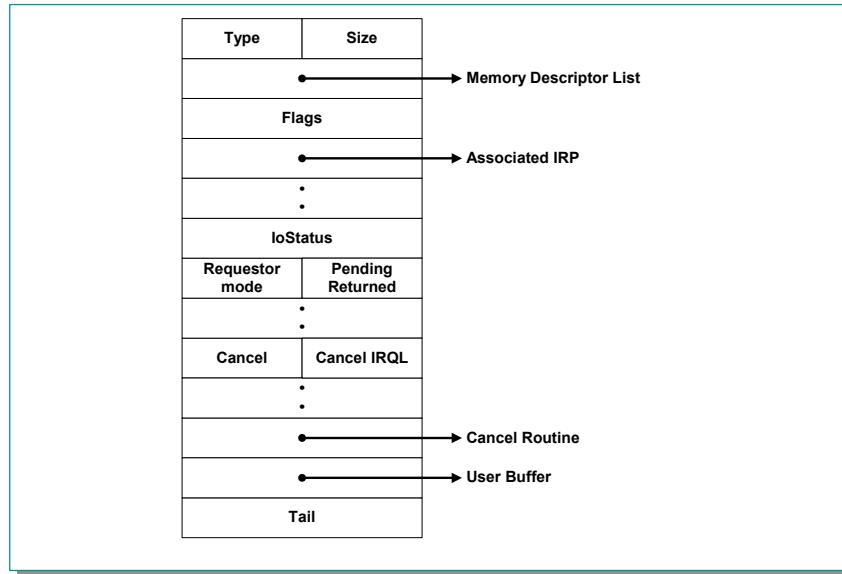
The IRP is where the I/O system stores information it needs to process an I/O request. When a thread calls an I/O service, the I/O manager constructs an IRP to represent the operation as it progresses through the I/O system. The I/O manager stores a pointer to the caller's file object in the IRP.

This slide shows the relationship between an IRP and the file, device, and driver objects described in the preceding sections. Although this example shows an I/O request to a single-layered device driver, most I/O operations are not this direct; they involve one or more layered drivers.

An IRP consists of two parts: a fixed portion (called a header) and one or more stack locations. The fixed portion contains information such as the type and size of the request, a pointer to a buffer for buffered I/O, and state information that changes as the request progresses. An IRP stack location contains a function code, function-specific parameters, and a pointer to the caller's file object.

While active, each IRP is stored in an IRP queue associated with the thread that requested the I/O. This arrangement allows the I/O system to find and free any outstanding IRPs if a thread terminates or is terminated with outstanding I/O requests. For more information on IRPs and the system routines that process them, see the DDK documentation.

I/O Request Packets



Key fields in an I/O Request Packet

MdlAddress

Points to an MDL describing a user buffer for an IRP_MJ_READ or IRP_MJ_WRITE request if the driver set up its device object(s) for direct I/O.

Flags

File system drivers use this field, which is read-only for all drivers.

AssociatedIrp.MasterIrp

Points to the master IRP in an IRP that was created by a highest-level driver's call to IoMakeAssociatedIrp.

AssociatedIrp.SystemBuffer

Points to a system-space buffer for one of the following: (1) a transfer request to a driver that set up its device object(s) requesting buffered I/O; (2) an IRP_MJ_DEVICE_CONTROL request, (3) an IRP_MJ_INTERNAL_DEVICE_CONTROL request with an I/O control code that was defined with METHOD_BUFFERED. In any case, the underlying device driver usually transfers data to or from this buffer.

IoStatus

Is the I/O status block in which a driver stores status and information before calling IoCompleteRequest.

RequestorMode

Indicates the execution mode of the original requestor of the operation, one of UserMode or KernelMode.

PendingReturned

If set to TRUE, a driver has marked the IRP pending. Each IoCompletion routine should check the value of this flag. If the flag is TRUE, and if the IoCompletion routine will not return STATUS_MORE_PROCESSING_REQUIRED, the routine should call IoMarkIrpPending to propagate the pending status to drivers above it in the device stack.

Cancel

If set to TRUE, the IRP either is or should be cancelled.

CancelIrql

Is the IRQL at which a driver is running when IoAcquireCancelSpinLock is called.

CancelRoutine

Is the entry point for a driver-supplied Cancel routine to be called if the IRP is cancelled. NULL indicates that the IRP is not currently cancelable.

UserBuffer

Contains the address of an output buffer if the major function code in the I/O stack location is IRP_MJ_INTERNAL_DEVICE_CONTROL and the I/O control code was defined with METHOD_NEITHER.

Tail.Overlay.DeviceQueueEntry

If IRPs are queued in the device queue associated with the driver's device object, this field links IRPs in the device queue. These links can be used only while the driver is processing the IRP.

Tail.Overlay.DriverContext

If IRPs are not queued in the device queue associated with the driver's device object, this field can be used by the driver to store up to four pointers. This field can be used only while the driver owns the IRP.

Tail.Overlay.Thread

Is a pointer to the caller's thread control block. Higher-level drivers that allocate IRPs for lower-level removable-media drivers must set this field in the IRPs they allocate. Otherwise, the FSD cannot determine which thread to notify if the underlying device driver indicates that the media requires verification.

Tail.Overlay.ListEntry

If a driver manages its own internal queue(s) of IRPs, it uses this field to link one IRP to the next. These links can be used only while the driver is holding the IRP in its queue or is processing the IRP.

Debugger Commands: !thread, !irp, !irpfind, !devobj, !drvobj

- **!thread** lists IRPs issued by the thread
- **!irp** formats an IRP
- **!irpfind** searches for IRPs
- **!devobj** formats a device object
- **!drvobj** formats a driver object

I/O devices and their drivers are frequently involved in debugging efforts. The debugger includes a variety of command for finding, and displaying details of, currently-outstanding I/O operations and I/O devices. Refer to the debugger documentation for complete details and examples of these commands.

!thread

The **!thread** command can display a list of the I/O request packets (IRPs) that have been issued by the thread, but are not yet complete.

!irp *IrPAddress*

The **!irp** command gives details of the IRP at a given address.

!irpfind

This command can search memory for IRPs based on any of several criteria. IRPs can be searched for based on device object address, file object address, thread address, or several other commonly needed items.

!devobj *DeviceObject*

Displays the indicated device object. This can be specified by address or by the device object name.

!drvobj *DriverObject*

Displays the indicated driver object. This can be specified by address or by the driver object name.

Module 4 Labs



Module 5: Crash Dump Analysis I

Module 5 Overview

- Kernel-Mode Error Handling
- Varieties of Kernel-Mode Dump Files
- Creating a Kernel-Mode Dump File
- Opening a Kernel-Mode Dump File
- Collecting Information from a Kernel-Mode Dump

Kernel-Mode Dump Files

Kernel-Mode Error Handling

- **Fatal errors are declared by calling KeBugCheckEx**
 - First argument is "bug check code"
 - Other four arguments are "bug check parameters"
- **Possible results:**
 - Display blue screen with bug check data
 - A kernel debugger (such as WinDbg or KD) can be contacted.
 - A memory dump file can be written.
 - The system can automatically reboot.
 - A memory dump file can be written, and the system can automatically reboot afterwards.

When a kernel-mode error occurs, the default behavior of Windows is to display the blue screen with bug check data.

However, there are several alternative behaviors that can be selected:

- A kernel debugger (such as WinDbg or KD) can be contacted.
- A memory dump file can be written.
- The system can automatically reboot.
- A memory dump file can be written, and the system can automatically reboot afterwards.

This section covers how to create and analyze a kernel-mode memory dump file. There are three different varieties of crash dump files. However, it should be remembered that no dump file can ever be as useful and versatile as a live kernel debugger attached to the system that has failed.

Varieties of Kernel-Mode Dump Files

- **Complete Memory Dump**
 - This file contains all the physical memory for the machine at the time of the fault.
- **Kernel Memory Dump**
 - This file contains all the memory in use by the kernel at the time of the crash.
- **Small Memory Dump (“Minidump”)**
 - This file contains bug check information, list of loaded drivers, the PRCB, EPROCESS,ETHREAD, and kernel-mode stack for the process and thread that failed.
 - This is sent to Microsoft if you allow it (Online Crash Analysis)

There are three kinds of kernel-mode crash dump files:

- Complete Memory Dump
- Kernel Memory Dump
- Small Memory Dump (“Minidump”)

The difference between these dump files is one of size. The *Complete Memory Dump* is the largest and contains the most information, the *Kernel Memory Dump* is somewhat smaller, and the *Small Memory Dump* is only 64 KB in size.

The advantage to the larger files is that, since they contain more information, they are more likely to help you find the cause of the crash. (Note, however, that no kernel-mode dump file can provide as much information as actually debugging the crash directly with a kernel debugger.)

The advantage of the smaller files is that they are smaller and written more quickly. Speed is often valuable; if you are running a server, you may want the server to reboot as quickly as possible after a crash, and the reboot will not take place until the dump file has been written.

On Windows XP and later versions of Windows, even if you have selected Complete Memory Dump or Kernel Memory Dump, a minidump will be created as well. After you reboot, a **Windows Error Reporting** popup appears and asks if you wish to send this minidump to Microsoft. If you agree, this dump will be sent, and will be used for Online Crash Analysis (OCA) – Microsoft’s analysis of existing bugs in Windows and in device drivers.

Complete Memory Dump

The *Complete Memory Dump* is the largest in size. This file contains all the physical memory for the machine at the time of the fault.

This dump file requires a pagefile on your boot drive that is at least as large as your main system memory: it should be able to hold a file whose size equals your entire RAM plus one megabyte.

The Complete Memory Dump file is written to `%SystemRoot%\Memory.dmp` by default.

If a second bug check occurs and another Complete Memory Dump (or Kernel Memory Dump) is created, the previous file will be overwritten.

Kernel Memory Dump

The *Kernel Memory Dump* contains all the memory in use by the kernel at the time of the crash.

This kind of dump file is significantly smaller than the Complete Memory Dump.

Typically, the dump file will be around one-third the size of the physical memory on the system.

This dump file will not include unallocated memory, or any memory allocated to user-mode applications. It only includes memory allocated to the Windows kernel and hardware abstraction level (HAL), as well as memory allocated to kernel-mode drivers and other kernel-mode programs.

For most purposes, this crash dump is the most useful. It is significantly smaller than the Complete Memory Dump, but it only omits those portions of memory that are unlikely to have been involved in the crash.

The Kernel Memory Dump file is written to `%SystemRoot%\Memory.dmp` by default.

If a second bug check occurs and another Kernel Memory Dump (or Complete Memory Dump) is created, the previous file will be overwritten.

Small Memory Dump (“Minidump”)

The *Small Memory Dump* is a much smaller file than the other two kinds of kernel-mode crash dumps. It is exactly 64 KB in size, and requires only 64 KB of pagefile space on the boot drive.

A small memory dump file includes the following:

- The bug check message and parameters, as well as other blue-screen data.
- The processor context (PRCB) for the processor that crashed.
- The process information and kernel context (EPROCESS) for the process that crashed.
- The thread information and kernel context (ETHREAD) for the thread that crashed.
- The kernel-mode call stack for the thread that crashed. If this is longer than 16 KB, only the topmost 16 KB will be included.
- A list of loaded drivers.

In Windows XP and Windows .NET Server, the following items are also included:

- A list of loaded modules and unloaded modules.
- The debugger data block. This contains basic debugging information about the system.
- Any additional memory pages that Windows identifies as being useful in debugging failures. This includes the data pages that the registers were pointing to when the crash occurred, and other pages specifically requested by the faulting component.
- (*Itanium processor only*) The backing store.

- (*Windows .NET Server only*) The Windows type — for example, "Professional" or "Server".

This kind of dump file can be useful when space is greatly limited. However, due to the limited amount of information included, errors that were not directly caused by the thread executing at time of crash may not be discovered by an analysis of this file.

Since this kind of dump file does not contain images of any executables residing in memory at the time of the crash, you may also need to set the executable image path if these executables turn out to be important.

If a second bug check occurs and a second Small Memory Dump file is created, the previous file will be preserved. Each additional file will be given a distinct name, which contains the date of the crash encoded in the filename. For example, *mini022900-01.dmp* is the first memory dump file generated on February 29, 2000. A list of all Small Memory Dump files is kept in the directory %SystemRoot%\Minidump.

Creating a Kernel-Mode Dump File

- **Enabling Kernel-Mode Dump File Creation**
 - Control Panel | System | Advanced | Startup and Recovery
 - Paging file on boot partition must be large enough to hold temporary copy of dump
- **Forcing the Creation of a Kernel-Mode Dump File**
 - **Creating a Kernel-Mode Dump File from the Debugger**
 - .crash (Generate Dump File)
 - **Creating a Kernel-Mode Dump File from the Keyboard**
 - HKLM\{Sys\CurrentCS\Services\i8042prt\Parameters
 - CrashOnCtrlScroll equal to REG_DWORD 0x1
 - <Right CTRL>+<Scroll Lock>+<Scroll Lock>

This section describes how kernel-mode dump files can be created.

Before creating a kernel-mode dump file, you should select the type of the dump and the file name and location for the dump file.

After the dump file has been enabled, there are two events, which can cause the creation of the dump file:

- The system can crash due to a kernel-mode error
- The user can deliberately force the creation of a kernel-mode dump file

Finally, if you are unsure whether a crash dump file has been created, you might wish to verify that a kernel-mode dump file has been written.

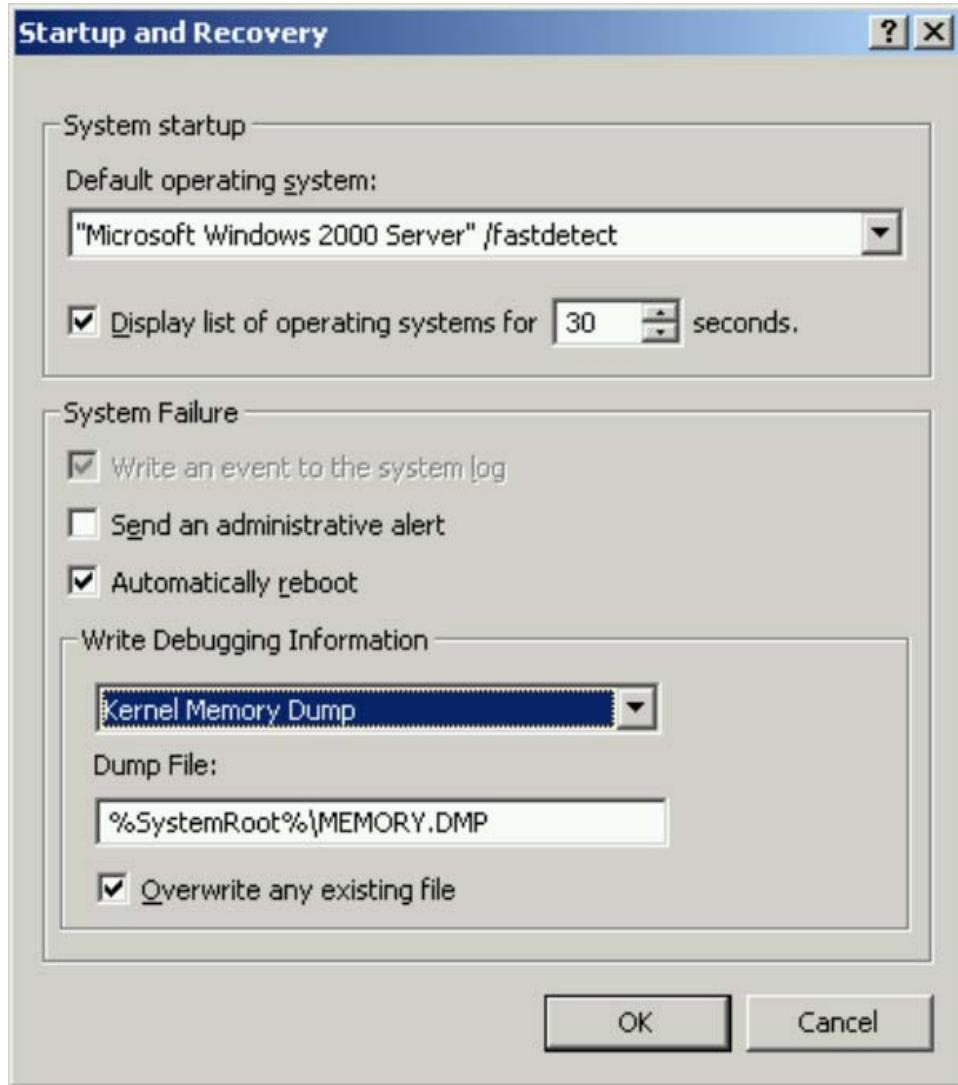
Enabling a Kernel-Mode Dump File

Before a crash dump is created, dump creation must be enabled and configured.

The Windows Control Panel controls the kernel-mode crash dump settings.

To change these settings, go to **Control Panel** and click on the **System** icon. Select the **Advanced** panel, and click on **Startup and Recovery**.

You will see the following dialog box:



Under Write Debugging Information, you can select which of the three sizes of dump files you wish to have as the default. Only one dump file can be created for any given crash. The default crash dump size is Small Memory Dump

After selecting the dump size, you can accept or change the default dump file path and file name.

You can also select or deselect any of the Write an event to the system log, Send an administrative alert, and automatically reboot options.

If you make any changes, the system will reboot after you click OK.

The settings that you select will apply to any kernel-mode dump file, whether it is created by a system crash, or by deliberately forcing the creation of the dump file.

Note that only a system administrator can modify these settings.

Forcing the Creation of a Kernel-Mode Dump File

Once kernel-mode dump files have been enabled, most system crashes should cause a crash file to be written and the blue screen to be displayed.

However, there are times that a system freezes without actually initiating a kernel crash. Possible symptoms of such a freeze include:

- The mouse pointer moves, but can't do anything.

- All video is frozen, the mouse pointer does not move, but paging continues.
- There is no response at all to the mouse or keyboard, and no use of the disk.

If an experienced debugging technician is present, he or she can hook up a kernel debugger and analyze the problem. For some tips on what to look for when this situation occurs, see Debugging a Stalled System.

However, if no technician is present, you may wish to create a kernel-mode dump file and send it to an off-site technician. This dump file can be used to analyze the cause of the error.

There are two ways to deliberately cause a kernel-mode crash dump file to be created:

- If a kernel debugger is attached, the crash can be initiated from the debugger.
- The crash can also be initiated from the keyboard if this has been enabled previously through the Registry.

Creating a Kernel-Mode Dump File from the Debugger

If WinDbg (or KD) is attached to a machine, it can force a crash dump to be created. This is done by entering the **.crash (Generate Dump File)** command at the kd> prompt. It may be necessary to follow this with the **G (Go)** command.

When this command is issued, the system will call **KeBugCheck** with a bug check code of **MANUALLY_INITIATED_CRASH** (0xE2). The crash dump file is written at this point.

After the crash dump file has been written, the kernel debugger on the host machine will be alerted and can be used to actively debug the crashed target.

Creating a Kernel-Mode Dump File from the Keyboard

A kernel-mode dump file can be created from the keyboard. This method is supported on most non-USB keyboards.

Two preparations must be made before this dump is created:

- The writing of crash dumps must be enabled, the path and file name chosen, and the size of the dump selected.
- Keyboard-initiated dumps must be enabled in the registry. In the registry path **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\i8042prt\Parameters**, set a key named **CrashOnCtrlScroll** equal to REG_DWORD 0x1. (Or any nonzero value.)

The system must be rebooted before these changes will take effect.

After this has been done, the keyboard crash can be initiated as follows. Hold down the rightmost CTRL key, and press the SCROLL LOCK key twice.

It is possible for a system to freeze in such a way that this CTRL+SCROLL LOCK+SCROLL LOCK sequence will not work. However, this should be a very rare occurrence. The CTRL+SCROLL LOCK+SCROLL LOCK crash initiation will work even in many instances where CTRL+ALT+DEL does not work.

The system then calls KeBugCheck with a bug check code of **MANUALLY_INITIATED_CRASH** (0xE2). The crash dump file is written at this point.

If a kernel debugger is attached to the frozen machine, the machine will break into the kernel debugger after the crash dump file has been written. However, in this case it is preferable to create the crash dump from the kernel debugger rather than the keyboard.

Analyzing a Kernel-Mode Dump File

Opening a Kernel-Mode Dump File

- Start WinDbg
- Open the dump file
- Resolve any symbol problems
- Use !analyze -v
 - Automatically interprets bug check code and parameters
 - For more details on a bug check, see the debugger documentation under “Bug Checks (Blue Screens)” → “Bug Check Code Reference”
 - You can inspect the stack trace given in the !analyze -v output for clues in module names and routine names

Starting the Session

You can begin the session as before:

- Start WinDbg
- Open the dump file
- Resolve any symbol problems

See Module 3 for details.

Using the !analyze -v Extension

The !analyze -v extension provides a first level of analysis for a crash dump. It will:

- Automatically interpret the bug check code and the bug check parameters
- Perform any necessary steps to set the debugger’s register context to that reflected in the crash dump file
- Display a stack trace with this register context in effect
- Display a “best guess” of which routine on the stack actually caused the failure

You should use the -v option for a fully verbose display of data. For details on other options, see the !analyze reference page in the debugger documentation.

A Kernel-Mode !analyze -v Example

In this example, the debugger is attached to a computer that has just crashed.

```
kd> !analyze -v
```

```
*****
*                               Bugcheck Analysis
*
*****
DRIVER_IRQL_NOT_LESS_OR_EQUAL (d1)
An attempt was made to access a pageable (or completely invalid) address
at an interrupt request level (IRQL) that is too high. This is usually
caused by drivers using improper addresses.
If kernel debugger is available get stack backtrace.
```

The first element of the display shows the bug check code and information about this type of bug check. Some of the text displayed may not apply to this specific instance. For more details on each bug check, see the debugger documentation.

Arguments:

```
Arg1: 00000004, memory referenced
Arg2: 00000002, IRQL
Arg3: 00000001, value 0 = read operation, 1 = write operation
Arg4: f832035c, address which referenced memory
```

The bug check parameters are displayed next. They are each followed by a description. For example, the third parameter is one, and the comment following it explains that this indicates that a write operation failed.

Debugging Details:

```
WRITE_ADDRESS: 00000004 Nonpaged pool
```

```
CURRENT_IRQL: 2
```

The next few fields vary depending on the nature of the crash. In this case, we see WRITE_ADDRESS and CURRENT_IRQL fields. These are simply restating the information shown in the bug check parameters. By comparing the statement "Nonpaged pool" to the bug check text that says "an attempt was made to access a pageable (or completely invalid) address," we can see that the address was invalid. The invalid address in this case was 0x00000004.

FAULTING_IP:

```
USBPORT!USBPORT_BadRequestFlush+7c
f832035c 894204      mov     [edx+0x4],eax
```

The FAULTING_IP field shows the instruction pointer at the time of the fault.

```
DEFAULT_BUCKET_ID: DRIVER_FAULT
```

The DEFAULT_BUCKET_ID field shows the general category of failures that this failure belongs to.

```
BUGCHECK_STR: 0xD1
```

The BUGCHECK_STR field shows the bug check code, which we have already seen. In some cases additional triage information is appended.

```
TRAP_FRAME: f8950dfc -- (.trap ffffffff8950dfc)
.trap ffffffff8950dfc
ErrCode = 00000002
eax=81cc86dc ebx=81cc80e0 ecx=81e55688 edx=00000000 esi=81cc8028
edi=8052cf3c
```

```

eip=f832035c esp=f8950e70 ebp=f8950e90 iopl=0          nv up ei pl nz ac
po nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000
efl=00010216
USBPORT!USBPORT_BadRequestFlush+7c:
f832035c 894204      mov     [edx+0x4],eax
ds:0023:00000004=?????????
.trap
Resetting default context

```

The TRAP_FRAME field shows the trap frame for this crash. This information can also be viewed by using the .trap command.

```
LAST_CONTROL_TRANSFER: from f83206e0 to f832035c
```

The LAST_CONTROL_TRANSFER field shows the last call on the stack. In this case, the code at address 0xF83206E0 called a function at 0xF832035C. You can use the LN command to determine what module and function these addresses reside in.

```

STACK_TEXT:
f8950e90 f83206e0 024c7262 00000000 f8950edc
USBPORT!USBPORT_BadRequestFlush+0x7c
f8950eb0 804f5561 81cc8644 81cc8028 6d9a2f30
USBPORT!USBPORT_DM_TimerDpc+0x10c
f8950fb4 804f5644 6e4be98e 00000000 ffdfff000 nt!KiTimerListExpire+0xf3
f8950fe0 8052c47c 8053cf20 00000000 00002e42 nt!KiTimerExpiration+0xb0
f8950ff4 8052c16a efdefd44 00000000 00000000 nt!KiRetireDpcList+0x31

```

The STACK_TEXT field shows a stack trace of the faulting component.

```

FOLLOWUP_IP:
USBPORT!USBPORT_BadRequestFlush+7c
f832035c 894204      mov     [edx+0x4],eax

```

The FOLLOWUP_IP field shows the disassembly of the instruction that has probably caused the error.

```
FOLLOWUP_NAME: usbtri
```

```
SYMBOL_NAME: USBPORT!USBPORT_BadRequestFlush+7c
```

```
MODULE_NAME: USBPORT
```

```
IMAGE_NAME: USBPORT.SYS
```

```
DEBUG_FLR_IMAGE_TIMESTAMP: 3b7d868b
```

The SYMBOL_NAME, MODULE_NAME, IMAGE_NAME, and DEBUG_FLR_IMAGE_TIMESTAMP fields show the symbol, module, image, and image timestamp corresponding to this instruction (if it is valid), or to the caller of this instruction (if it is not).

```
STACK_COMMAND: .trap ffffffff8950dfc ; kb
```

The STACK_COMMAND field shows the command that was used to obtain the STACK_TEXT. You can use this command to repeat this stack trace display, or alter it to obtain related stack information.

```
BUCKET_ID: 0xD1_W_USBPORT!USBPORT_BadRequestFlush+7c
```

The BUCKET_ID field shows the specific category of failures that the current failure belongs to. This category helps the debugger determine what other information to display in the analysis output.

INTERNAL_SOLUTION_TEXT:

<http://oca.microsoft.com/resredir.asp?sid=62&State=1>

If you are connected to the internet, the debugger attempts to access a database of crash solutions maintained by Microsoft®. This database contains links to a tremendous number of webpages that have information on known bugs. If a match is found for your problem, the INTERNAL_SOLUTION_TEXT field will show a URL that you can access for more information.

Followup: usbtri

This problem has a known fix.

Please connect to the following URL for details:

<http://oca.microsoft.com/resredir.asp?sid=62&State=1>

There are a variety of other fields that may appear:

- If control was transferred to an invalid address, then the FAULTING_IP field will contain this invalid address. Instead of the FOLLOWUP_IP field, the FAILED_INSTRUCTION_ADDRESS field will show the disassembled code from this address, although this disassembly will probably be meaningless. In this situation, the SYMBOL_NAME, MODULE_NAME, IMAGE_NAME, and DBG_FLR_IMAGE_TIMESTAMP fields will refer to the caller of this instruction.
- If the processor misfires, you may see the SINGLE_BIT_ERROR, TWO_BIT_ERROR, or POSSIBLE_INVALID_CONTROL_TRANSFER fields.
- If a bug check occurred within the code of a device driver, its name may be displayed in the BUGCHECKING_DRIVER field.

The Followup Field and the *trage.ini* File

In both user mode and kernel mode, the Followup field in the display will show information about the owner of the current stack frame, if this can be determined. This information is determined in the following manner:

When the !analyze extension is used, the debugger begins with the top frame in the stack and determines whether it is responsible for the error. If it isn't, the next frame is analyzed. This process continues until a frame that might be at fault is found.

The debugger attempts to determine the owner of the module and function in this frame. If the owner can be determined, this frame is considered to be at fault.

If the owner cannot be determined, the debugger passes to the next stack frame, and so on, until the owner is determined (or the stack is completely examined). The first frame whose owner is found in this search is considered to be at fault. If the stack is exhausted without any information being found, no Followup field is displayed.

The owner of the frame at fault is displayed in the Followup field. If !analyze -v is used, the FOLLOWUP_IP, SYMBOL_NAME, MODULE_NAME, IMAGE_NAME, and DBG_FLR_IMAGE_TIMESTAMP fields will refer to this frame.

For the Followup field to display useful information, you must first create a *trage.ini* file containing the names of the module and function owners.

The *triage.ini* file should identify the owners of all modules that could possibly have errors. You can use an informational string instead of an actual "owner," but this string cannot contain spaces. If you are certain that a module will not fault, you can omit this module or indicate that it should be skipped. It is also possible to specify owners of individual functions, giving the triage process an even finer granularity.

For details on the syntax of the *triage.ini* file, see the debugger documentation.

Collecting Information from a Kernel-Mode Dump File

- **.bugcheck** – displays bug check code and parameters
- **!process 0 0** – lists all processes with their image names
- **!process -** displays information about a specified (or current) process
- **!thread -** displays information about a specified (or current) thread
- **!drivers** – displays list of each driver loaded
- **!locks** – displays kernel eresource locks
- **!vm** – displays a summary of memory use on the target system
- **!pool** – displays information about suspected pool address
- **!timer** – displays the system timer list
- **!errlog** - displays the contents of any pending error log entries

In many cases it is impossible to precisely determine the reason for a crash. It is valuable to collect as much information from the crash dump as possible. This information can be used to group various “indeterminate” crash dumps into sets based on the processes that are running, the drivers that are loaded, and so on. This can help to identify common factors in a set of crash dumps.

We have described most of these commands in detail already; the following descriptions focus on the usefulness of this command sequence in collecting information from a crash dump:

.bugcheck

The most basic information about any crash dump is the bug check code and the four bug check parameters. Most software problems tend to produce the same bug check code, or at least the same small set of bug check codes, over and over. Hardware problems on the other hand, particularly memory, CPU, and bus errors, frequently create a set of crashes with widely varying bug check codes. The **.bugcheck** command displays the bug check code and parameters.

!process 0 0

We use this command to produce a list of the executable images that are running on the system.

!thread

The **!thread** command, used with no parameters, will display information about the currently executing thread. We can then use **!process** on the indicated “Owning Process” address to learn which process in the system owns that thread.

!drivers

A list of the drivers present in the system is often an essential clue. If a set of “indeterminate” crashes have a particular driver as a common factor, that driver should be examined closely.

!locks

Systems that have “frozen” in various ways often do so due to deadlocks associated with *executive resource locks*. The **!locks** command will display all of the locks held by all threads. The thread displays from **!process** will then show which threads are waiting on the held locks.

!lvm

Some system “hangs” are caused by low resources, particularly in the areas of physical memory, virtual memory (paging file space), and the paged and nonpaged pools. Check the output of **!lvm** for any unusually high values.

!pool Address

The **!pool** extension must be supplied the address of a memory location within a region allocated from pool. It will then display the pool regions “around” the indicated address, checking for anomalies such as corrupted headers. If your debugging procedures provide you with any addresses that you suspect might lie within the pool, using the **!pool** extension on such addresses may help to identify problems.

!timer

The **!timer** extension displays all of the pending kernel timer requests in the system. Many of these are commonly associated with device drivers. Timer requests are a way in which driver code runs other than as a result of an I/O request from an application or an interrupt from a device. The system timer request list should therefore be examined for any signs of pending operations by suspect drivers.

!errlog

Displays the contents of any pending error log entries.

This command displays information about any pending events in the I/O system's error log. These are events queued by calls to the **IoWriteErrorLogEntry** function, to be written to the system's event log for subsequent viewing by the **Event Viewer**.

Only entries that were queued by **IoWriteErrorLogEntry**, but have not been committed to the error log, will be displayed.

This command can be used as a diagnostic aid after a system crashes because it reveals pending error information that was unable to be committed to the error log before the system halted.

Module 5 Labs



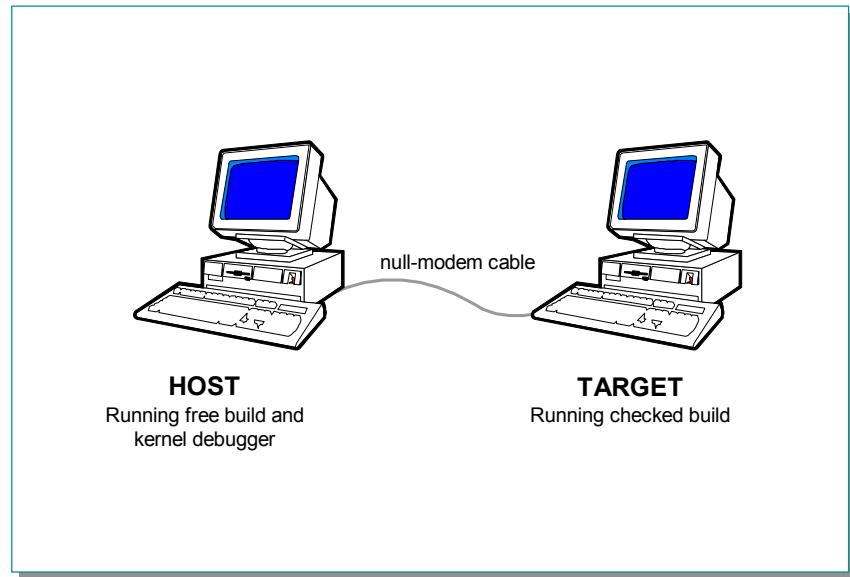
Module 6: Kernel Debugging I

Module 6 Overview

- Setup for kernel mode live debugging
- Live kernel debugger operations

Setting Up a Kernel Debugging Session

Configuring Hardware for Kernel-Mode Debugging



Target Computer and Host Computer

Kernel-mode debugging requires a *target computer* and a *host computer*. The target computer is used to run the kernel-mode program. The host computer is used to run the debugger.

Typical Debugging Setup

This slide illustrates the "typical" setup. However, the current versions of KD and WinDbg, which have been installed with this documentation, are extremely flexible:

- They can debug a target computer that is running Windows NT® 4.0, Windows 2000, or Windows XP, or Windows .NET Server 2003.
- They can debug a target computer that is running on an x86 platform, an Itanium platform, or an AMD x86-64 platform.
- They can be launched from a host computer that is running Windows NT 4.0, Windows 2000, or Windows XP, or Windows .NET Server.
- They can be launched from a host computer that is running on an x86 platform or an Itanium platform.

The target computer and host computer do not have to be using the same platform or the same version of Windows.

Kernel debugging does not require specific combinations of the free or checked builds. It is possible to debug a free system from a free or checked system or to debug a checked system from a free or checked system. However, there is generally no reason for the host computer to run the slower checked build.

If you are running the debuggers from an Itanium or AMD x86-64 host computer, you may need the 64-bit debugger binaries. See the debugger documentation for details.

Cables Used for Debugging

- **Null-Modem Cable (also called “Debug Cable”)**
 - Attaches to COM port
 - Default speed is 19200 bps
 - Maximum speed is 115200 bps
- **1394 Cable (also called “Firewire cable” or “iLink”)**
 - Attaches to 1394 port
 - Extremely fast

When the host computer is at the same location as the target computer, or when you need to put a local host computer with remote access server (RAS) capabilities between the target and a remote host, the two computers must be connected with either debug (null-modem) cables or 1394 (fire-wire) cables.

Setting Up a Null-modem Cable Connection

Null-modem cables are serial cables that have been configured to send data between two serial ports. They are available at most computer stores. Do not confuse null-modem cables with standard serial cables, which do not connect serial ports.

If you are dialing straight into the target system’s modem, debugging a failed user-mode process on the same computer, or analyzing a dump file, a null-modem cable is not needed.

The default serial port for debug output from the target computer is the highest enumerated port (usually COM2). This can be changed by setting the **debugport** option.

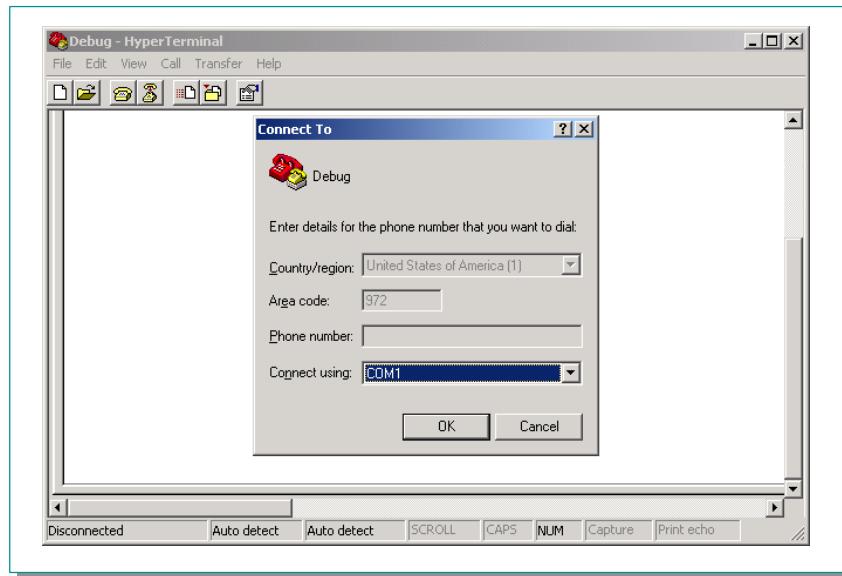
The default serial port for debug input to the host computer is COM1. This can be changed by setting the **_NT_DEBUG_PORT** environment variable.

Setting Up a 1394 Cable Connection

Kernel debugging through a 1394 cable is not supported on all systems. The target computer must be running either Windows XP or Windows .NET Server 2003, and the host computer must be running one of these systems as well (not necessarily the same Windows version as the host, however).

The target computer and host computer will each have a 1394 adapter. Plug the 1394 cable into any port on the target computer, and into any port on the host computer. The choice of port does not matter, and it does not affect the channel number that will be used during software setup.

Testing the Connection



In the event the debugger is unable to attach to the target computer, the cable connection between the host and target computers should be confirmed as valid.

Testing a Null-Modem Cable Connection

The following procedure will apply to both the host and the target computers:

1. Click the **Start** button, point to **Programs**, then point to **Accessories**, then point to **Communications**, then click **HyperTerminal**. If HyperTerminal is not installed, install it from the product CD-ROM using **Add/Remove Programs** from Control Panel.
2. In the **Connection Description** dialog box, enter a name for the new connection ("debug" is acceptable – you can use the same connection name on both computers).
3. The **Connect To** dialog box is next. In the Connect using drop down list box, select the COM port corresponding to the port to which the null-modem cable is connected on that machine.
4. Accept the defaults for the COM port properties in the next dialog box.
5. Once this procedure has been completed on both systems, HyperTerminal will be open and ready for testing. Type in a string of characters on the host computer. If the null-modem cable is properly installed and the correct COM ports were chosen within HyperTerminal on both systems, the string of characters typed in on the host will be displayed in the HyperTerminal window of the target computer. Be sure to note the specific COM port used on each machine. You will need this information later.

If the keystrokes on the host do not appear on the target computer, confirm the cable is snugly plugged into both systems. Also be sure that the cable is a null-modem cable as opposed to a pass-through serial cable.

If the cable checks out, then the likely problem is with the COM ports. Create a new connection in HyperTerminal on the target computer using an alternative COM port following steps two through five and retest. If the problem is not resolved, try changing the COM port on the host computer. If the problem persists, change the target computer's

COM port setting back to the original setting and retest. Eventually the correct configuration will be made and the test will succeed.

Configuring the Target Computer

- Configure *boot.ini* file -- add a line where Windows is "debugger enabled"
- You can have several lines containing different Windows versions or different options
- If you don't edit *boot.ini*, you can still select debugging mode with F8 boot options screen



The target computer must be configured to enable kernel-mode debugging prior to suffering a bug check. This is done with a simple edit to the *boot.ini* file on an x86 target computer.

Typically this configuration change is not implemented until a recurring problem arises, as the configuration permanently reserves one COM port for the debugger. However, for systems that must always be available for interactive debugging, this configuration change can be implemented at any time.

Once kernel debugging is enabled through a COM port or a 1394 port, Windows removes this port from the system device list. The Control Panel utilities will either not list this port at all, or will show "Not enough resources to use this port". This does not represent any malfunction. However, if you wish to test the null-modem connection using HyperTerminal or another communication program, you must do so before enabling debugging.

When kernel debugging is enabled for an operating system on a dual-boot machine, "[debugger enabled]" will be displayed in the description of that operating system in the boot menu.

Configuring a Target Computer for Debugging

The kernel debugger is enabled in the *boot.ini* file on an x86 computer. You must edit this file to activate the target computer for kernel debugging in case of a bug check.

To configure a target computer for a remote or local remote debugging, edit the *boot.ini* file by adding the appropriate debugger options to notify Windows to load the kernel debugger. The *boot.ini* file is located in the root directory of the system partition (usually drive C).

Configuring the Target Computer (continued)

- **Editing *boot.ini* for a null-modem cable**
 - `/debugport=COMn /baudrate=Speed`
 - If you omit `/baudrate`, the default speed is 19200 bps
 - You should never omit `/debugport` -- don't rely on default!
- **Editing *boot.ini* for a 1394 cable**
 - `/debugport=1394 /channel=Channel`
 - `Channel` specifies 1394 channel number -- any unused channel, from 1 through 62

Debugger Boot Options

To configure the target computer for kernel debugging, add the `/debug` option to the *boot.ini* file. If you are not going to use the default COM port or the default baud rate, you will need to use additional options. There is also an option that enables the kernel debugger only upon system stops, which frees up the COM port during normal use. However, using this option prevents a debugger tool from gaining access to the target computer during normal operation.

The following table lists the boot options that can be used to configure an x86 target computer for debugging:

Option	Description
<code>/nodebug</code>	Disables kernel debugging (even for FSE's). This is the default. If you specify <code>/nodebug</code> , then <code>/debugport</code> , <code>/baudrate</code> , and <code>/crashdebug</code> are ignored.
<code>/debug</code>	Causes the kernel debugger to be loaded during boot and kept in memory at all times. This means that a support engineer can dial into the system being debugged and break into the debugger, even when the system is not suspended at a Stop message.
<code>/debugport=Port</code>	Specifies the serial port to be used by the kernel debugger. If no serial port is specified, the debugger defaults to COM2.
<code>/crashdebug</code>	Causes the kernel debugger to be loaded during boot but swapped out to the paging file after boot. As a result, a support engineer cannot break into the debugger unless Windows is suspended at a Stop message.
<code>/baudrate=BaudRate</code>	Sets the speed that the kernel debugger uses in bits per second (bps). The default rate is 19,200 bps. 9,600 bps is the normal rate for remote debugging over a modem. The higher baud rates (greater than 57600KB) can be unreliable.

/kernel	Use this switch to specify a different kernel file than the one located in the default location, typically c:\winnt\system32. This can be useful in testing Hot Fixes and other patches to the kernel before making a permanent change to the system.
/hal	You use this switch to specify a different hardware abstraction level (HAL) file than the one located in the default location, typically c:\winnt\system32. This can be useful in testing Hot Fixes and other patches to the HAL before making a permanent change to the system.
/maxmem=SizeInMB	Specifies the amount of memory to be made available to the system. For example, you could restrict a 128 MB computer to 32 MB.
/sos	Causes the system loader to display the names of the drivers as they load when Windows NT is booting.

Note When you use the **/debugport** or **/baudrate** options, you do not need to use the **/debug** option, as Windows assumes that the computer will start in debug mode. You must use at least one of the options described in the table above to configure a computer for remote debugging. Otherwise, Windows does not load the kernel debugger.

Builds with **/debug** should show up on the boot menu with "[debugger enabled]" at the end of the description.

To Set the Startup Options in the *boot.ini* File

1. Start Windows Explorer and select the root folder of drive C.
2. Right-click *boot.ini* (or *boot*, if file extensions are hidden). Select **Properties**, and remove the read-only attribute of this file. Click **OK**.
3. Double-click the file icon to start Notepad with *boot.ini* loaded. (Any text editor will work.) The *boot.ini* file will look similar to this:

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINNT
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINNT="Microsoft Windows 2000
Professional" /fastdetect /debug /baudrate=19200
```

Note: This sample is from an x86 computer running Windows 2000 Professional.

4. Select the startup option that you normally use (line five in this example) and add the **/debug** option at the end of the line if no other debugging options are to be used.
5. If necessary, use the option **/debugport=comx** at the end of the same line to specify the communications port, where *x* is the communications port that you want to use.
6. If necessary, use the option **/baudrate=9600** at the end of the same line to specify the data communication rate. This would only be necessary in the case of a direct dial connection using a modem running on an x86 system.

This is the result of the change to the sample *boot.ini* file after steps one through six have modified it:

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINNT
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINNT="Microsoft Windows 2000
Professional" /fastdetect /debugport=com2
```

7. Save the *boot.ini* file and quit Notepad.
8. Shutdown the computer and restart Windows.
9. At boot menu select “Debugging Enabled” option

Example

For an x86 computer, the following line in *boot.ini* would boot Windows 2000 in a simulated 512 MB environment, with kernel debugging enabled on COM1, and communicating at 115200 baud:

```
multi(0)disk(0)rdisk(0)partition(1)\WINNT="Microsoft Windows 2000
Professional" /fastdetect /MAXMEM=512 /DEBUGPORT=COM1 /BAUDRATE=115200
```

Resolving Boot Problems

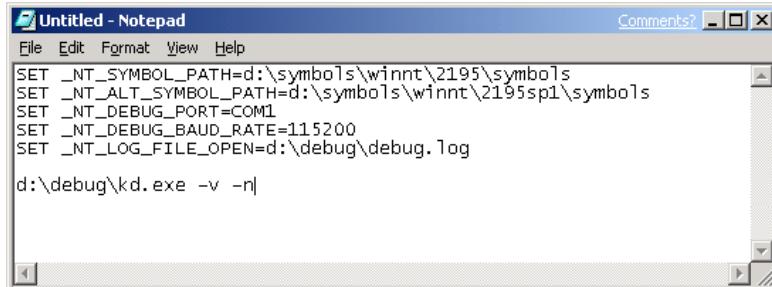
If the target computer consistently generates a Stop message during the startup process, or does not keep running long enough for you to edit the *boot.ini* file to enable the kernel debugger, you can try these options:

- **If the target computer is running Windows 2000 or a later version of Windows:** Early in the process of starting Windows, a character-mode menu displays the operating system choices. Press the F8 key at that time to display the **Advanced Options Menu**, and choose to start the computer in Safe Mode. Safe Mode loads only the minimum required drivers and system services during the startup of Windows. If the computer does successfully startup in Safe Mode, you can use Notepad to edit *boot.ini*.
- If your boot partition is formatted as file allocation table (FAT), you can start MS-DOS® from a bootable floppy disk and use a text editor to edit *boot.ini*. You can also copy the file to a floppy disk, take it to another computer to make the edits, and copy the modified version over the original on the target computer.
- If your boot partition is formatted as FAT32, you can use a Startup floppy disk from Windows 95 or Windows 98 and use a text editor to edit *boot.ini*. You can also copy the file to a floppy disk, take it to another computer to make the edits, and copy the modified version over the original on the target computer. (Since the *boot.ini* file is hidden and read-only, you will have to change these attributes before you can edit it. You can see the file by using the **dir /ah** command, and change the file's attributes by issuing an **attrib -r -s -h boot.ini** command.)
- If your boot partition is formatted as NTFS file system, you have two choices.
 1. Restart your computer from either of the three Windows installation floppy disks or the product CD-ROM. Once you get to the Welcome screen in the Windows setup, press the F10 key to start the Recovery Console tool. The Recovery Console is a troubleshooting tool that enables access to NTFS disks from a command prompt. From the Recovery Console prompt, type **SET AllowRemovableMedia = TRUE** and press **Enter**. Use the **Copy** command to copy the *boot.ini* to a floppy disk. Take the disk to another computer to make the necessary edit, then use **Copy** command again to copy the modified version over the original on the target computer.

2. You can also install a second copy of Windows on a different partition and start the computer from the new installation. (You cannot gain access to files on an NTFS partition from MS-DOS, Windows 95, or Windows 98.) Then you can use Notepad to edit the original installation's *boot.ini*.
 - If you previously created a Windows boot recovery disk for the computer that has the problem, you can use this disk on another computer to edit the *boot.ini* file, and then start the target computer.

Configuring Software on the Host Computer

- Configure Environment Variables for WinDbg or KD
- Avoids need for command line options



```
Untitled - Notepad
File Edit Format View Help
Comments? [ ] X
SET _NT_SYMBOL_PATH=d:\symbols\winnt\2195\symbols
SET _NT_ALT_SYMBOL_PATH=d:\symbols\winnt\2195sp1\symbols
SET _NT_DEBUG_PORT=COM1
SET _NT_DEBUG_BAUD_RATE=115200
SET _NT_LOG_FILE_OPEN=d:\debug\debug.log
d:\debug\kd.exe -v -n
```

Before you can begin kernel debugging, you must install all necessary symbol files, and set certain environment variables.

Installing Symbol Files

The symbol files matching the operating system installation of the target computer must be installed on the host computer.

Configuring Environment Variables

The kernel debugger uses a variety of environment variables to indicate a number of important settings.

If you will be working with the kernel debugger on a regular basis, you might want to create a batch file that includes all of the environment variables, which you commonly use.

The following is an example of such a batch file:

```
SET _NT_SYMBOL_PATH=d:\symbols\winnt\2195\symbols
SET _NT_ALT_SYMBOL_PATH=d:\symbols\winnt\2195sp1\symbols
SET _NT_DEBUG_PORT=COM1
SET _NT_DEBUG_BAUD_RATE=115200
SET _NT_LOG_FILE_OPEN=d:\debug\debug.log

d:\debug\kd.exe -v -n
```

The Kernel Debugging Session

Beginning the Debugging Session

- **Using a Debug Cable**
 - `windbg OtherOptions -k com:port=Port,baud=Rate`
- **Using a 1394 Cable**
 - `windbg OtherOptions -k 1394:channel=Channel`
- **Local Kernel Debugging (on one machine)**
 - `windbg OtherOptions -kl`
- **WinDbg Menu**
 - File | Kernel Debug | COM
 - File | Kernel Debug | 1394
 - File | Kernel Debug | Local

Before you can begin kernel debugging with WinDbg, you must specify exactly how the connection to the target computer is to be made.

The symbol path must be set before kernel debugging can be successful.

WinDbg is also capable of performing *local kernel debugging*: in other words, debugging the operating system of the machine on which the debugger itself is running. Local kernel debugging is only supported on Windows XP and later versions of Windows.

WinDbg Command Line

Beginning a kernel debugging session from the WinDbg command line is done in the exact same way as in KD.

```
windbg [-y SymbolPath] -k com:port=Port,baud=Rate
windbg [-y SymbolPath] -k 1394:channel=Channel
windbg [-y SymbolPath] -k serial:modem
windbg [-y SymbolPath] -kl
windbg [-y SymbolPath]
```

Note that the full WinDbg command-line syntax is not the same as that of KD.

WinDbg Menu

When WinDbg is in dormant mode, you can begin a kernel debugging session by selecting the **File | Kernel Debug** menu command or by pressing the CTRL+K shortcut key.

When the **Kernel Debugging** dialog box appears, select the **COM** tab, the **1394** tab, or the **Local** tab. Each of them specifies a different connection method:

- The **COM** tab indicates that the connection will use a COM port. In the **Baud Rate** text box, enter the baud rate. In the **Port** text box, enter the name of the COM port. (This can be in the format "com2" or in the format "\\.\com2", but should not simply be a number). Then click **OK**. If you select **OK** without entering text, WinDbg checks to see if the _NT_DEBUG_PORT and _NT_DEBUG_BAUD_RATE environment variables have been set. If not, WinDbg defaults to COM1 and 19200 baud.
- The **1394** tab indicates that the connection will use 1394. In the **Channel** text box, enter the 1394 channel used at boot on the target computer. If you do not specify the channel, WinDbg checks to see if the _NT_DEBUG_1394_CHANNEL environment variable has been set.
- The **Local** tab indicates that WinDbg will perform local kernel debugging.

Beginning the Session

If the target computer has crashed, the target computer is still halted as a result of an earlier kernel debugging action, or the **-b** command-line option was used, then the debugger will break into the target computer immediately.

Otherwise, the target computer will continue running until the debugger orders it to break.

Breaking Into the Target Computer

- **When the Target is Running – Stop Target With**
 - WinDbg: CTRL+Break , Debug | Break, “pause” button
 - KD: CTRL+C
- **When the Target is Stopped**
 - Debugger displays current instruction
 - Most debugger commands described so far will work
 - Exceptions are !analyze, .bugcheck, and other “crash dump related” commands
 - **Use “G” (go) command, or F5, or “run” button to resume normal execution on target**

While debugging a target application in user mode or target computer in kernel mode, there are two basic states. The target can be *running* or the target can be *stopped*.

When the debugger first connects to a user-mode target, it will immediately stop the target, unless the **-g** command-line option was used.

When the debugger connects to a kernel-mode target, it will leave the target running, unless the **-b** command-line option was used, or the target computer has already crashed, or the target computer is still halted as a result of an earlier kernel debugging action.

When the Target is Running

When the target is running, most debugger actions are unavailable.

If you wish to stop a running target, you can issue a **Break** command. This causes the debugger to *break into the target*: in other words, it will stop the target and all control will be given to the debugger. The program may not break immediately. For example, if all threads are currently executing system code, or are in a wait operation, the program will break only after control has returned to the program's code.

If a running target encounters an exception, if certain events occur, if a breakpoint is hit, or if the program terminates normally, the target will *break into the debugger*. This will stop the target and give all control to the debugger. A message will be displayed in the Debugger Command window describing the error, event, or breakpoint.

When the Target is Stopped

A variety of commands exist to start or control the target's execution:

- To cause the application to begin running, issue the **Go** command.
- To step through the program one instruction at a time, use the **Step Into** or **Step Over** commands. If a function call occurs, **Step Into** will enter the function and continue stepping through each instruction, while **Step Over** will treat the function call as a single step. When the debugger is in Assembly Mode, stepping is done one machine instruction at a time. When the debugger is in Source Mode, stepping is done one source line at a time.

- To finish the current function and stop when the return occurs, use the **Step Out** or **Trace and Watch** commands. The **Step Out** command will continue until the current function ends. **Trace and Watch** will do the same, and also display a summary of that function's calls; however, the **Trace and Watch** command has to be issued on the first instruction of the function in question.
- If an exception has occurred, the **Go with Exception Handled** and **Go with Exception Not Handled** commands can be used to resume execution and control the status of the exception.
- (*WinDbg only*) If you select a line in the Disassembly window or a Source window and then use the **Run to Cursor** command, the program will execute until this line is reached.
- (*User Mode only*) To terminate the target application and restart it from the beginning, use the **Restart** command. This command can only be used with a process which was originally created by the debugger. After the process is restarted it immediately breaks into the debugger.
- (*Kernel Mode only*) To reboot the target computer, use the **Reboot** command.
- (*WinDbg only*) To terminate the target application and clear the debugger, use the **Terminate Target** command. This allows you to begin debugging a different target.

Examining the Target Computer's State

• When The Target Is Stopped

- Registers window (ALT+4) permits modification as well as display
- Memory window permits modification as well as display of memory
- The “E” command can modify contents of memory
- Use stack display commands (KV, etc.) to see recent call history
- Use !thread, !process, to see which thread of which process the CPU is “in”

Debugger Operation

When the target system is stopped, most of the debugger commands we have described already will work. There are a number of additional capabilities due to the fact that we are looking at an active (although temporarily stopped) computer rather than a “snapshot” of a crashed system’s memory.

Registers and Memory Windows

The Registers and Memory windows permit the modification of the displayed values.

If you are doing live debugging, registers are also automatically displayed each time the target halts. This means that if you are stepping through your code with the **P (Program Step)** or **T (Trace)** commands, you will see a register display at every step. To stop this display, use the **r** option with these commands.

E (enter data) command

This command may be used in the command window to modify the contents of memory.

KV (stack backtrace, verbose) command

As usual this command will display the stack of the current thread. The routine name and offset at the top of the stack indicate the address of the instruction that will be executed *next*, when the target system is allowed to resume execution.

!thread, !process

When you “break into” a target system with the kernel debugger, you might have interrupted literally any thread within any process in the system. Use these commands to see what thread in what process, running what image was executing at the time you stopped the system.

Synchronizing with the Target Computer

- Sometimes during kernel-mode debugging, the target machine will stop responding to the debugger.
- To fix this, use **CTRL+ALT+R** or **Debug | Kernel | Connection | Resynchronize**

Sometimes during kernel-mode debugging, the target computer will stop responding to the debugger.

In KD, use the **CTRL+R (Re-synchronize)** key followed by Enter to synchronize with the target computer.

In WinDbg, use **CTRL+ALT+R** or **Debug | Kernel | Connection | Resynchronize**.

This will often restore communication between the host and the target.

Ending the Debugging Session

- **Exiting WinDbg**

- **Q (Quit)**
- **File | Exit**

Ending the Session but not Exiting

When performing user-mode debugging in WinDbg, you can end the debugging session without exiting WinDbg. This is done by selecting the **Debug | Stop Debugging** menu command. This can also be done by pressing the SHIFT+F5 shortcut key or by using the following button from the toolbar:



You will be prompted to save the workspace for the current session, and then WinDbg will return to dormant mode. At this point, all starting options are available to you: you can begin debugging a running process, spawn a new process, attach to a target computer, open a crash dump, or connect to a remote debugging session.

Exiting WinDbg

When you are using WinDbg to do kernel debugging, there are two ways to exit.

Issuing a **Q (Quit)** command in WinDbg will save the log file, end the debugging session, and exit the debugger. The target computer will remain locked.

Selecting the **File | Exit** menu command, or using the ALT+F4 shortcut key, will allow you to exit the debugger even if there is no command prompt. This allows you to let the target computer keep running after the debugger is terminated.

Module 6 Labs



Module 7: Understanding Disassembled Code

Module 7 Overview

- **x86 Architecture**
 - **x86 Instructions**
 - **Annotated x86 Disassembly**
-

x86 Architecture

CPU Architecture

- Pentium i-386 CISC (x86) vs. Itanium (IA-64, VLIW/EPIC) vs. Opteron (AMD x86-64) vs. Alpha RISC vs. ...
- A processor architecture includes:
 - Address and Data Widths
 - Registers
 - General purpose, flags, privileged, floating point, ...
 - Instructions
 - Data movement, arithmetic, logical, string, flow control, ...
 - Operand types
 - Memory management mechanisms
- General Disassembly Syntax:
 [Prefix] Instruction [operand 1],[operand 2]

Processor Architecture

The term *processor architecture* refers to the way a processor is configured, the registers and other memory locations on the processor, and the machine instructions used by that processor.

Almost all the examples in this documentation are taken from an **x86 processor** (also known as Pentium or i-386). The architecture on these chips is 32-bit CISC (Complex Instruction Set Computer).

WinDbg can also debug the **Itanium processor** (whose architecture is sometimes referred to as “IA-64”) and the **AMD x86-64 processor** (whose architecture is sometimes referred to as “AMD64”).

Current versions of Microsoft Windows do not run on the **Alpha processor**. The architecture on these chips is RISC (Reduced Instruction Set Computer).

Registers and Flags

Each processor has a number of general-purpose registers. Registers are special-purpose memory locations that are located in the microprocessor itself. Accessing registers is much faster than accessing external memory. Some registers are general purpose, while others have dedicated functions.

General-purpose registers are used to hold values for computation and store results internally. Registers in general are byte, word, or DWORD addressable.

The flags register is the only bit addressable register. Any instructions affect the flags register based on the results of a computation. For example, after a subtraction instruction, the zero flag is set if the result is zero and cleared if the result is non-zero.

Instruction Notation

Assembly instructions take zero, one, or two arguments called operands. An operand can be an immediate value, register value, or a pointer to a memory location.

The general syntax for Intel Assembly is the following:

Prefix *Instruction* [*Operand 1*], [*Operand 2*] ; *Comment*

<i>Prefix</i>	An address label or instruction modifier. <i>Prefix</i> is often omitted.
<i>Instruction</i>	The instruction being executed
<i>Operand 1</i>	Typically the destination operand, but it can be the source operand (one argument instruction), or it can be both the source and destination operands (INC instruction).
<i>Operand 2</i>	Source operand for two argument instructions.
<i>Comment</i>	Any additional text. This is ignored by the assembler.

For example, consider the instruction **MOV EAX,[12341234]**. The brackets indicate that the address is being “dereferenced.” Thus the source operand is the value at address 0x12341234. The destination operation is the EAX register. So this instruction will move the value stored at [12341234] into the EAX register.

Arithmetic instructions are typically two-register with the source and destination registers combining. The result is stored into the destination.

To save space, many of the instructions are expressed in combined form; which means that the first parameter must be a register, but the second can be a register or a memory reference or an immediate value.

```
MOV    EAX, 177
ADD    EAX, EDX
```

To save even more space, instructions can also be expressed with the first parameter as a register or a memory reference, and the second can be a register, memory reference, or immediate value.

Unless otherwise noted, when this abbreviation is used, you cannot choose "memory" for both source and destination

Disassembling Backwards

On the x86 processor, instructions are variable-sized; so disassembling backwards is an exercise in pattern matching. To disassemble backwards from an address, you should start disassembling at a point farther back than you really want to go, then look forward until the instructions start making sense. The first few instructions may not make any sense because you may have started disassembling in the middle of an instruction. There is a possibility, unfortunately, that the disassembly will never synchronize with the instruction stream and you will have to try disassembling at a different starting point until you find a starting point that works.

Intel CPU Registers

- **i-386 Microprocessors have 6 General-Purpose Registers**
 - EAX, EBX, ECX, EDX, EDI, and ESI
 - First two bytes and first word of EAX, EBX, ECX, EDX are individually addressable
 - AL – first byte of EAX
 - AH – second byte of EAX
 - AX – first word of EAX
 - First word of EDI and ESI are addressable (DI, SI)
- **Non-Orthogonal Machines**
 - Only some instructions can be used with certain registers

Intel 386+ processors contain a variety of 32-bit registers. The i-386 family of processors is known as *non-orthogonal*. This means the register and the instruction set are not completely interchangeable. In other words, some instructions can only be used with certain registers. For example, IN and OUT instructions are hardwired to the accumulator (EAX) register, ECX is used for counting for loop instructions, and EDI and ESI are used for indexing/string instructions. Other addressing modes can only be applied to certain registers.

The i-386 family has six general-purpose registers – EAX, EBX, ECX, EDX, EDI, ESI. Each register begins with the letter “E” for “Extended” to address the full 32-bits instead of just 16.

Addressing

EAX, EBX, ECX, EDX registers can be addressed in the following ways:

AL	lower 8 bits of the EAX register
AH	high order byte of lower order word of the EAX register
AX	lower 16 bits of the EAX register
EAX	the full 32 bits of the register

Similarly syntax exists for EBX, ECX, and EDX.

Note:

The ESI and EDI registers are not byte addressable.

Register Types

- **General-Purpose**
 - EAX = Holds value returned by called functions
 - ECX = Used for counting during REP (repeat) operations
 - EDX = Holds high 32 bits of value returned by functions returning 64-bit values
 - EDI, ESI = used by block move and compare instructions
- **Stack – ESP, EBP**
- **Flags Register – EFLAGS**
- **Special Purpose Registers – EIP**

General-Purpose Registers

EAX is sometimes referred to as the *Accumulator*. Many instructions return results in this register, even if the instruction does not name the register explicitly. It is common for compiled code that returns a value to do so using the EAX register.

ECX specializes in counting. Certain instructions use this register to indicate the number of operations performed before the instruction completed, or to limit the number of operations that may be performed. However it can also be used for many purposes not associated with these instructions.

EBX and EDX are general-purpose registers and are used for addressing memory as a pointer, used as operands for logic and arithmetic functions, and contain the results of instructions. EAX and ECX can also be used for these functions.

Indexing Registers

EDI, ESI are general-purpose registers that specialize in indexing. String instructions use EDI as the destination pointer, and ESI for the source pointer. For example, to copy a block of memory from one place to another, ESI would be set up to point to the source block, and EDI would be set up to point to the destination block. ECX would be loaded with the number of bytes to transfer, the direction flag would be set up to increment or decrement, and then REP MOVS would copy the bytes.

Stack Registers

ESP and EBP are primarily used for stack manipulation and control. ESP is the stack pointer and used to point to the current stack location, that is, to the data most recently pushed onto the stack. The EBP register is used to point to the stack frame for a given routine. At routine entry, the EBP register is typically saved on the stack and then set equal to the value of the current stack pointer. The EBP is then used to reference arguments and local variables.

Flag Register

The flags register is a collection of single-bit flags. Many instructions alter the flags to describe the result of the instruction. These flags can then be tested by conditional jump instructions.

Instruction Pointer

EIP is the instruction pointer. It points to the next instruction to be executed. EIP is modified by RET, RETI, JMP, CALL, and INT instructions. This register is perhaps the most important register in the CPU since it directs the CPU to the next instruction to execute.

x86 Instructions

Operands

- **Implicit**
 - CLI – Clear Interrupt Flag
STI – Set Interrupt Flag
- **Register**
 - MOV EAX,EBX ; copies contents of EBX register to EAX.
; EAX<-EBX EAX = destination EBX = source
- **Immediate**
 - MOV EAX,177 ; load EAX register with 177
- **I/O**
 - IN AL,04H ; read port location 4 into AL register
- **Memory Reference...**

Assembly instructions each take a variable number of arguments called operands. This varies on the type of instruction and varies from zero to two operands.

Implicit Operands

Implicit operands are specified by the instruction itself. CLI and STI are examples of instructions that do not specify an operand. The CLI and STI clear and set the interrupt flag in the EFLAGS register. CLI and STI always apply to that bit only.

Register Operands

Register operands are operands in which the instruction references a register for the source or destination. Some instructions reference a single register – which can be either, or both the source and destination.

Example:

```
CALL EDI ; EDI contains address to call
MOV EAX,EBX ; EAX<-EBX EAX = destination EBX = source
INC EAX ; EAX=EAX+1 - EAX=source and destination
```

Immediate Operands

Immediate operands are part of the instruction itself. The value to be used follows the opcode in the instruction. Immediate operands are frequently used to load a constant into a variable or to reference a hard-coded or fixed-memory location.

Example:

```
MOV    EAX,177      ; load EAX with 177
MOV    CL,0FFH       ; load CL with 255
CALL   12341234H    ; call routine at location 12341234H
```

I/O Operands

I/O operands are not used in user mode programs. I/O instructions are used to reference devices in the I/O map. These include most hardware devices like PICS, serial ports, parallel ports, disk controllers, etc.

Example:

```
IN     AL,04H        ; read port location 4 into AL register
OUT   04H,AL         ; write AL register to port location 4
```

Memory Reference Operands

Used to Reference Memory via Different Addressing Modes:

- | | |
|-------------------------------|-----------------------|
| • Direct | MOV AL, [12341234H] |
| • Based | MOV AL, [ECX] |
| • Base + Displacement | MOV ECX, [EBP-0x10] |
| • Index + Displacement | MOV EAX,1234124H[ESI] |
| • Base + Displacement + Index | MOV EAX,[EBP+8][ESI] |

Memory reference operands are used to reference memory via a multitude of addressing modes. These addressing modes include direct addressing, based addressing, base plus displacement addressing, index plus displacement addressing, and base plus displacement plus index addressing.

Direct Addressing

Direct addressing is where the address itself is used as an operand. This is used to address fixed locations in a program – a global variable for instance.

Example:

```
MOV AL, [12341234H]  
INC DWORD PTR [12341234H]
```

Based Addressing

Based addressing uses a register to hold the address. This is used to de-reference a pointer to a variable.

Example:

```
MOV AL, [ECX]  
DEC DWORD PTR [ESI]
```

Base Plus Displacement Addressing

Base Plus Displacement Addressing is similar to based addressing except for an additional offset that is added to the base address. This is used to access a variable contained in a structure. For example, the base (register) can point to the start of the structure, and the offset allows the correct element to be referenced. This is better than the pointer having to point directly to the element since often more than one structure element is referenced by a section of code. This way the base is set once and only the offset changes.

Example:

```
LEA    EDX, [ESP+0x4]
MOV    ECX, [EBP-0x10]
```

Index Plus Displacement Addressing

Index Plus Displacement Addressing is similar to base plus displacement except the register and fixed offset have reversed rolls. Now the base address is a fixed location in memory plus an offset contained in a register. This is primarily used to access elements of an array declared in static memory.

Example:

```
MOV    EAX, 1234124H[ESI]
```

Base Plus Displacement Plus Index Addressing

Base Plus Displacement Plus Index Addressing is the combination of addressing modes. It combines the elements of the previous two methods. It is used to address arrays declared off the stack or dynamically out of the heap. A register points to the base, another register contains an offset, and a fixed offset can also be included.

Example:

```
MOV    EAX, [EBP+8][ESI]
INC    WORD PTR [EBX+EAX*2]
```

Accessing Memory

Global/Static Variables

MOV EAX, [00401234]	load EAX from value stored at 00401234
MOV EAX, 0	load EAX immediately with 0
LEA ESI, copyright_msg	load effective address into ESI

Local Variables

MOV EAX, [EBP-0x20]	load EAX with local variable
---------------------	------------------------------

Dynamically Allocated Variables (Heap)

LEA ESI, _m_pFileList	load pointer
MOV EAX, [ESI]	read value

Just like any high level language, Assembly can access variables in a number of different ways. There are three basic storage classes for variables:

Global/static – allocated out of programs data section

Local variables/Arguments – allocated off the stack

Heap variables – allocated off the heap

Global Static

Global static variables are located at a fixed address (at least to the program they are fixed addresses). The most common way to access these variables is to use the fixed address in the instruction itself.

```
MOV EAX, [12341234H] ; loads EAX with value stored
                      ; at location 12341234H
INC DWORD PTR TEST2!_nCount ; increments DWORD variable
                            ; nCount
```

The debugger will use symbolic information when available.

Local Variables/Arguments

Local variables/arguments reside on the stack and are referenced using the EBP and sometimes the ESP. Optimized code will often eliminate the need for a frame pointer; in these cases the ESP register is used to access local variables and EBP can be used for an additional general-purpose register. When a standard stack frame is used you will see instructions like these:

```
MOV EAX, [EBP+8] ; load EAX with argument
MOV EAX, [EBP-4] ; load EAX with local variable
```

The trick to remember is that most of the time (when EBP is not being used as a general purpose register), if the offset is positive, an argument is being accessed. If offset is

negative then a local variable is being accessed. The first argument passed to a function is typically EBP+8.

Heap allocated variables reside in the heap, therefore they are accessed via a pointer. Typically more than one instruction is required.

```
MOV    ESI, TEST2! m_pFileList ; load the pointer  
MOV    EAX, [Esi+4] ; read second DWORD (pszName) in heap
```

Another thing to consider is that most compilers will move commonly accessed variables into a register for faster access.

Arithmetic Instructions

ADD	EAX,55	EAX = EAX + 55
SUB	EAX,1	EAX = EAX - 1
MUL	EAX,21	EAX = EAX * 21
DIV	EAX,4	EAX = EAX / 4 EDX = Remainder
SHL	EAX,4	shift EAX by four bits
INC	EAX	increment EAX by 1
DEC	EAX	decrement EAX by 1
NEG	EAX	negate (2's complement)

The standard arithmetic operations are also available – ADD, SUB, INC, DEC, MUL, and DIV.

Shift and rotate operations are available.

Often compilers will use these instructions in odd-looking sequences that execute faster than a more obvious combination. Compilers often do complex evaluations of instruction ordering to achieve the fastest possible execution speeds. Not all instructions operate at the same rate – the number of instructions per second varies depending on the instructions and the addressing modes used.

```
XOR    EAX, EAX      ; is faster than MOV EAX, 0
SHL    EAX, 3         ; is faster than multiplying EAX by 8.
SHR    EAX, 3         ; is faster than dividing EAX by 8.
```

Logic Instructions

OR	EAX,1	set low bit of EAX
AND	EAX,0FFH	clear all but the low byte of EAX
XOR	EAX,1	toggle low bit of EAX
XOR	EAX,EAX	clears EAX to zero

Logic Instructions

The standard bitwise operations are available – AND, OR, XOR, and NOT. These operate all at the bit level. (A byte has 8-bits, so EAX which is 4 bytes, for instance, is 32-bits in length.)

As an interesting special case, XOR of a register with itself results in that register being cleared to zero. This is the fastest way to generate a zero, and you'll see it often in compiled code.

String Instructions

REP STOSD set memory block to DW value in EAX
REPNE SCASB scan string for value in EAX

MOVS – Move string
CMPS – Compare string
STOS – Store accumulator
LODS – Load accumulator
SCAS – Scan String

String and memory instructions are designed to manipulate blocks of memory with a single instruction. Registers are dedicated to specific purposes:

EAX – used as an accumulator
ECX – used as a counter
EDI – used as a destination pointer
ESI – used as a source pointer

Examples include:

MOVS – Move string
CMPS – Compare string
STOS – store accumulator
LODS – load accumulator
SCAS – Scan String

String instructions when prefixed with REP (repeat) will use ECX as a counter.

Stack Instructions

- **PUSH EAX**
 - subtracts four from ESP, then copies EAX contents to location pointed to by ESP
- **POP EAX**
 - copies memory pointed to by ESP to EAX, then adds four to ESP
- **ADD ESP, nn**
 - “pops” *nn* bytes off of the stack without copying them anywhere

PUSH and POP are the basic stack manipulation instructions. Their operation is illustrated in detail on the following pages.

It is frequently useful to “remove” things from the stack without actually copying the data “removed” anywhere – for example, when temporary storage has been allocated on the stack and is of no further use or consequence. In such an instance, the usual implementation is to simply add the size (in bytes) of the area to be removed to the stack pointer register ESP. Note that the data “removed” is still in memory at the same addresses. However it will be overwritten by subsequent stack PUSH operations.

Thread Stacks

- Two stacks for every thread
 - One for user mode (1 MB, pageable)
 - One for kernel mode (12 KB, normally nonpageable)
- ESP register (stack pointer) always contains address of last item pushed onto stack
- "r" Command – shows ESP register (among other)

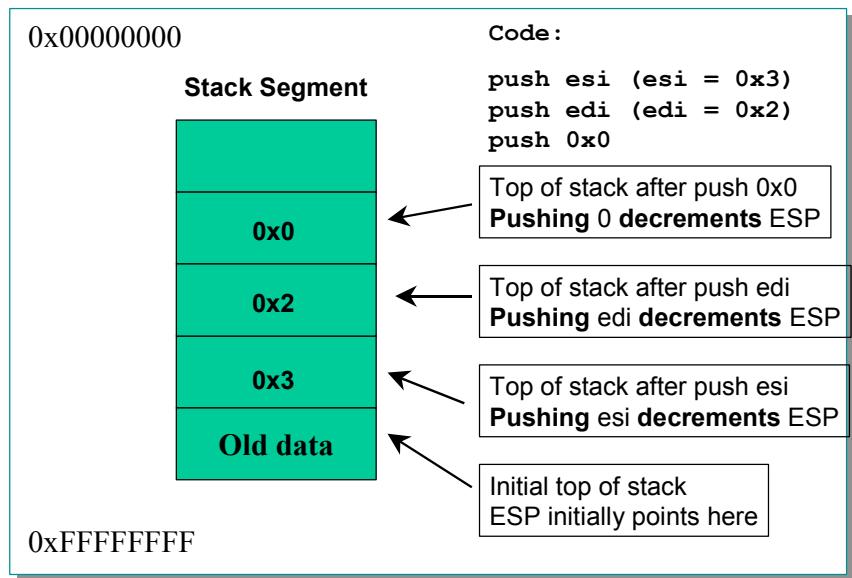
```
kd> r
EAX=01b40000 EBX=7ffd000 ECX=00000000
EDX=00000001 ESI=00142851 EDI=00000000
EIP=01b4211c ESP=0012ff18 EBP=0012ffc0
EFL=00000206 CS=001b DS=0023 ES=0023 SS=0023
FS=0038 GS=0000
```

Having discussed basic stack operations, the focus is now on the specifics of stack creation and operation in Win32. When a thread is created, one megabyte of virtual memory is reserved for use by the thread as a stack. (There are actually two thread stacks: a kernel-mode stack and a user-mode stack.)

The ESP register is set to point to the top of the user-mode stack (in other words, the highest address), and the stack is initialized. To examine the current value of the stack pointer, use the "r" command in the debugger. The ESP register holds the current value of the stack pointer.

```
> r
EAX=01b40000 EBX=7ffd000 ECX=00000000 EDX=00000001
ESI=00142851 EDI=00000000
EIP=01b4211c ESP=0012ff18 EBP=0012ffc0 EFL=00000206
CS=001b DS=0023 ES=0023 SS=0023 FS=0038 GS=0000
```

Pushing Items onto the Stack



A computer stack operates like a stack of plates. Items are always added and removed from the current top of the stack. In this plate example, multiple plates could be picked up at one time in order to access plates lower on the stack. Once plates are removed, the new uppermost plate is now the top of the stack.

Adding items to a computer stack is referred to as "pushing" an item. The x86 instruction for pushing an item onto a stack is called the PUSH instruction. Pushing an item onto a computer stack causes the current top of the stack to be decremented by four bytes before the item is placed on the stack.

When plates are added to the stack, all the previously added plates move downwards and the new plate sits at the top of the stack. A computer stack would be very inefficient if it had to move all the previously pushed items in memory every time a new value was pushed. To avoid this inefficiency, a pointer is used to keep track of where the top of the stack is currently located. This pointer is stored in the ESP register and aptly named the Stack Pointer. When items are pushed onto a stack, the following events happen.

1. The value of the Stack Pointer (ESP) is decremented by four bytes.
2. The pushed item is moved into the stack address pointed to by ESP.

Since the value of the stack pointer is decremented with each PUSH instruction, the stack grows downwards in memory. This is an important point to remember for later stack analysis.

Popping Items Off the Stack

When an item is popped from the stack, the value at the current top of stack is retrieved and ESP is incremented by 4 bytes.

Code:

```
pop eax  
pop edi  
pop esi
```

Result:

```
eax = 0x0  
edi = 0x2  
esi = 0x3
```

Stack Segment

0x0
0x2
0x3
Old Data

Initial value of ESP
Value of ESP after first pop
Value of ESP after second pop
Final value of ESP

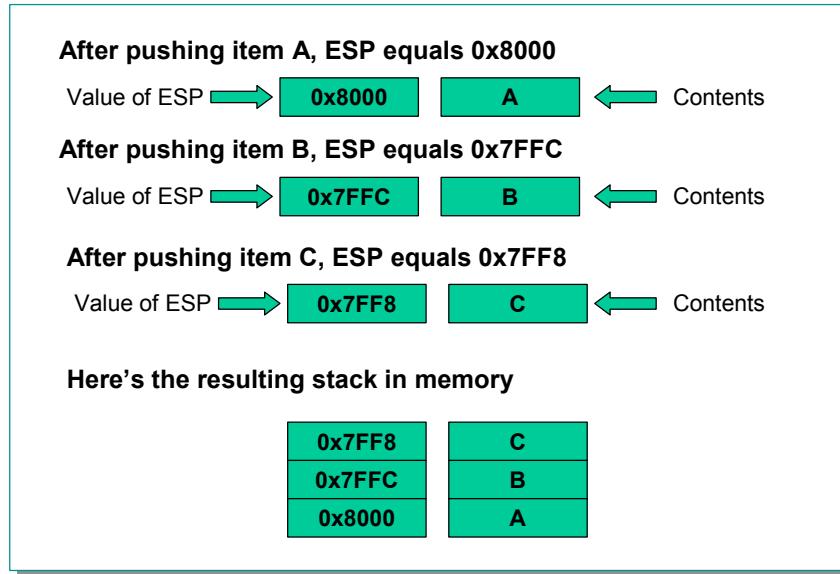
As items are needed, they are picked up from the top of the stack.

To retrieve an item from the stack, the POP instruction is issued. The order of operations for a POP instruction is the reverse of a PUSH.

1. The item pointed to by the Stack Pointer is retrieved from the stack.
2. The Stack Pointer is incremented by four bytes.

This causes the Stack Pointer to always point to the next available item on the stack.

Pushing Items Using a Stack Pointer



Using the example in the slide, the value of the Stack Pointer held in ESP is shown along with the item at that stack location. Assume the following assembly instructions are being executed.

```
PUSH A
PUSH B
PUSH C
```

Initially, the value of ESP is 0x8004, which means the Stack Pointer is pointing to memory address 0x8004. When the first PUSH instruction is executed, the value of the Stack Pointer is decremented by four bytes and the Stack Pointer now equals 0x8000. The value A is stored in memory location 0x8000. Next, the value B is to be pushed onto the stack. The Stack Pointer is again decremented by four bytes and now equals 0x7FFC and B is stored at this location. This continues until the final value C is stored in memory location 0x7FF8.

Flow Control Instructions

Unconditional	
NOP	no operation
JMP TEST2!_directory+2c	jump direct to label
JMP DWORD PTR [EBP + 12]	jump indirect
INT 2e	software interrupt
CALL TEST2!_Search	call subroutine
CALL DWORD PTR [EBX+EAX*4]	call indirect
RET 4	return
Conditional	
CMP EAX, EBX	sets flags according to EAX-EBX
TEST EAX, EBX	sets flags according to EAX AND EBX
JNZ loop_again	jump if zero flag not set
LOOP loop_again	loop until ECX = 0

Flow control instructions are either unconditional or conditional. The latter occur only if a particular condition (specified by the instruction) is met. These statements support high level language constructions like procedures, if-then-else, switch case, etc.

Unconditional Instructions

JMP Operand

This instruction simply sets EIP to the operand following the instruction. Nothing is saved on the stack and no flags are set. JMP is used for fixed branches in execution; most if-then-else clauses require a minimum of one JMP instruction.

CALL Operand

This instruction first saves EIP on the stack, then sets EIP to the operand. Pushing EIP allows the program to return to the instruction following the CALL statement after the called procedure is complete.

For both the JMP and CALL instructions the operand can be a fixed address, a register value, or a pointer to the address to branch to (indirect jmp or call).

RET Operand

The RET instruction puts the current value on the stack into the EIP register. The operand is used for fixing up the stack pointer for arguments passed on the stack. This is used by the _stdcall and _pascal calling conventions. C calling convention requires the calling procedure to fixup the stack.

INT Operand

The INT instruction causes a software interrupt where the operand is the interrupt number. This is similar to the call instruction except the EFLAGS register is also pushed onto the stack. If this is performed in user mode, a switch to kernel mode also occurs. Interrupt routines end with a RETI instruction – this pops the EFLAGS as well as the EIP.

Conditional Instructions

LOOP Address

The LOOP instruction is used to implement loops in high-level languages. It will branch to the specified “address” until the ECX register is zero. If ECX is not zero, ECX is decremented and the loop continues.

```
XOR    EAX,EAX      ; clear EAX register
MOV    ECX, 5        ; load loop count

START:
ADD    EAX,1          ; add one to eax
LOOP   START
```

JNZ, JE, etc. Address

The jump-on-condition instructions jump based on the state of a condition. For example, JNZ (jump not zero) will jump to the specified address if the ZERO flag is not set. These instructions are primarily used for if-clauses.

```
XOR    EAX,EAX      ; clear eax
MOV    ECX,5

START:
ADD    EAX,1 ; add one to EAX
DEC    ECX   ; decrement loop counter
JNZ   START
```

TEST Operand1, Operand2

CMP Operand1, Operand2

These are not conditional flow control instructions in and of themselves, but are commonly used in combination with such instructions.

TEST is another way of doing an AND except it doesn’t copy the result to the destination register. It does, however, set condition flags – for example, if the result of a TEST is 0 then the Zero flag is set. TEST using the same value for both operands (such as TEST EAX, EAX) is often used as a way of seeing if a value is 0 or not, and in this instance is usually followed by JZ or JNZ.

Similarly, CMP is a subtract that doesn’t store its result anywhere. It subtracts Operand2 from Operand1, and then sets the condition flags according to the result. For example, if Operand1 is less than Operand2, the sign flag will be set to indicate that the result of the most recent operation was negative. A variety of jump-on-condition instructions exist to then implement Jump if less than, jump if less than or equal, etc., in both signed and unsigned variations.

CALL and RET Instructions

- **CALL *entryPointName***
 - PUSHes the address of the next instruction after the CALL (the return address) onto the stack
 - transfers control (jumps) to *entryPointName*
- **RET**
 - POPs the saved return address off of the stack and transfers control there
- **RET n**
 - Same as RET, but after POPping the return address, adds n to stack pointer

These are the fundamental instructions used in calling, and returning from, procedures. We will describe them in great detail in the next module.

Annotated x86 Disassembly

Relating C to Assembly

```
#include <stdio.h>
int Addemup(int,int);
void main(void)
{
    int x = 5, y = 10, z = 0;
    z = Addemup(x,y);
    printf("z= %i\n",z);
}

int Addemup(int a, int b)
{
    int c = 0;
    c=a+b;
    return(c);
}
```

The following section will walk you through a disassembly example.

Taking the simple example we will look at the code generated by the compiler.

The Source Code

```
#include <stdio.h>

int Addemup(int,int);

void main(void)
{
    int x = 5;
    int y = 10;
    int z = 0;

    z = Addemup(x,y);

    printf("z= %i\n",z);
}

int Addemup(int a, int b)
{
    int c = 0;

    c=a+b;
    return(c);
}
```

The Annotated Assembly Code

```

1:      #include <stdio.h>
2:
3:      int Addemup(int,int);
4:
5:      void main(void)
6:      {
7:
8:          int x = 5;
9:          int y = 10;
10:         int z = 0;
11:
12:         z = Addemup(x,y);
13:
14:     }
15:
16:     int Addemup(int a, int b)
17:     {
18:
19:         int c = 0;
20:         c = a + b;
21:
22:         return(c);
23:
24:     }

```

addemup!main:

00401000 55	push	ebp	; save base pointer
00401001 8bec	mov	ebp,esp	; set stack pointer
00401003 83ec0c	sub	esp,0xc	; make room for locals

In this block we are pushing the frame pointer (stored in ebp) to the stack so we can unwind from this function successfully, then we are setting the stack pointer to the base pointer (ebp).

At this point the function can access its parameters as positive offsets from ebp, and the local variables as negative offsets.

00401006 c745fc05000000	mov	dword ptr [ebp-0x4],0x5	; local x = 5
0040100d c745f80a000000	mov	dword ptr [ebp-0x8],0xa	; local y = 10
00401014 c745f400000000	mov	dword ptr [ebp-0xc],0x0	; local z = 0

10:

0040101b 8b45f8	mov	eax,[ebp-0x8]	; load eax with y
0040101e 50	push	eax	; push y on stack
0040101f 8b4dfc	mov	ecx,[ebp-0x4]	; load ecx with x
00401022 51	push	ecx	; push x on stack
00401023 e81b000000	call	addemup!Addemup	; call Addemup
00401028 83c408	add	esp,0x8	; fixup stack for args
0040102b 8945f4	mov	[ebp-0xc],eax	; z returned in eax

12:

0040102e 8b55f4	mov	edx,[ebp-0xc]	; load edx with z
00401031 52	push	edx	; push z on the stack
00401032 6830704000	push	0x407030	; push ptr to "z=%i\n"
00401037 e822000000	call	addemup!printf	; call printf
0040103c 83c408	add	esp,0x8	; fix stack for args

14:

0040103f 8be5	mov	esp,ebp	; restore stack ptr
00401041 5d	pop	ebp	; restore base ptr
00401042 c3	ret		; return

15:

00401043 55	push	ebp	; save base pointer
00401044 8bec	mov	ebp,esp	; set stack pointer
00401046 51	push	ecx	; make room for local

18:

00401047 c745fc00000000	mov	dword ptr [ebp-0x4],0x0	; local c = 0
0040104e 8b4508	mov	eax,[ebp+0x8]	; set eax to local a
00401051 03450c	add	eax,[ebp+0xc]	; eax = eax + local b
00401054 8945fc	mov	[ebp-0x4],eax	; local c = a + b

21:

00401057 8b45fc	mov	eax,[ebp-0x4]	; set eax to result c
-----------------	-----	---------------	-----------------------

```
22:      }

0040105a 8be5          mov    esp,ebp      ; restore stack ptr
0040105c 5d             pop   ebp        ; restore base ptr
0040105d c3             ret            ; return to caller
```

Module 7 Labs



Module 8: Call Stacks

Module 8 Overview

- **Calling Conventions**
- **Analyzing the Stack**
- **Frame Pointer Omission**

This section defines and describes the use of the stack.

What You Will Learn

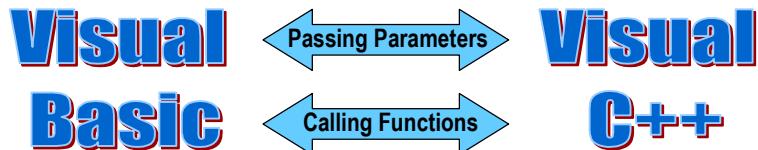
After completing this lesson, the student will be able to:

- List the information that must be obtained from a stack in order to analyze it
- Explain the purpose of calling conventions
- Describe how parameters are passed for a given calling convention
- Define prologue and epilogue code generation
- List several uses for a stack
- Describe how functions use stacks for temporary storage
- Display a call stack
- Define Frame Pointer Omission (FPO)
- Describe Frame Pointer Type Information
- Describe FPO Processes

Calling Conventions

Understanding Calling Conventions

- Guidelines for passing parameters and calling functions between languages due to varying architectures, compilers, and languages.
- STDCALL, CDECL, FASTCALL, etc.



If all programs were written in the same language using the same compiler, calling conventions would probably not exist as we know them today. However, with the variety of languages, compilers, and architectures, past and present, standardized methods for handling the different languages arose. In order for programmers to handle mixing these programming languages together, certain guidelines exist that specify how functions are called and parameters to these functions are passed. These rules for parameter passing are known as calling conventions.

The main use of a calling convention is to define how parameters are passed into a function. When a program written in C needs to access a function written in Pascal, the C programmer needs to know how Pascal expects the parameters to be handed to the Pascal function.

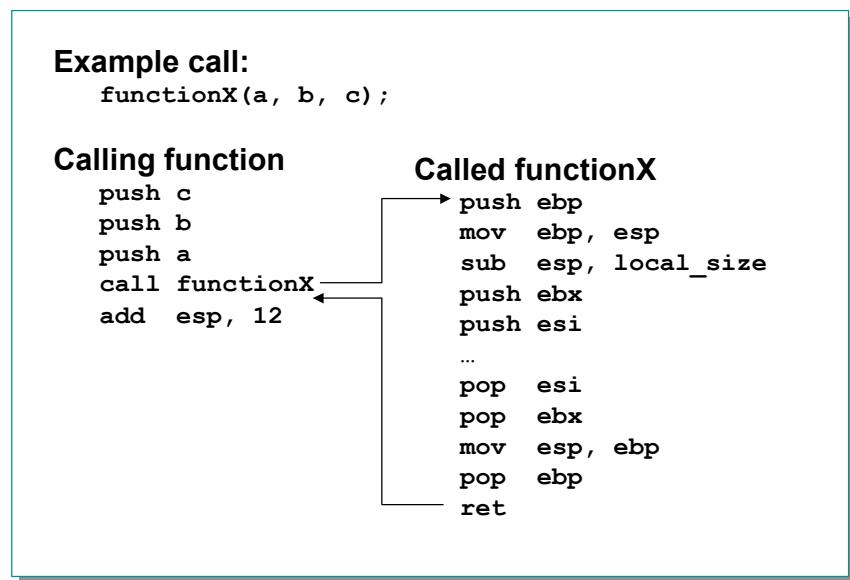
Example:

A programmer writes a function in Pascal called Sqrt(). This function takes a floating-point (real) value and returns the square root of the value passed as a real.

```
function Sqrt( NumArgument : real ) : real;
```

The C programmer needs to know how the Pascal programmer expects the variable "NumArgument" to be passed to the Sqrt function. To handle this mixed language programming, the Pascal language defined the Pascal calling convention. This convention is now known as the STDCALL calling convention and the term PASCAL is obsolete.

Calling Convention Example



Whenever a procedure is to be called, certain operations must be performed:

- The arguments, if any, to the procedure must be made available in a standardized way, usually on the stack.
- Control must be transferred to the called procedure, and the return address – the address to which control will be returned when the called procedure is complete – must be saved where it can be found later.
- The called procedure must save and restore any registers it is going to modify.
- The called procedure must allocate memory for variables declared locally to that procedure, in such a way that multiple invocations of the procedure (perhaps due to recursion) do not use the same areas of memory for their respective local variables.
- Upon return from the called procedure, all effects except the documented results of the procedure must be “undone.” Arguments, if any, must be removed from the stack; any registers modified must be restored to their original contents; and so on.

Procedure calling conventions define a standardized mechanism for performing these steps. By learning to recognize the various calling conventions that are used, and by learning how to interpret stack contents based on the calling conventions in use by a particular call, we can obtain a great deal more information from the stack than a simple list of nested procedure calls.

Prologue and Epilogue Code

- **Prologue Code**
 - The first instructions in a called procedure
 - Sets up the frame pointer
 - Reserves room on the stack for local variables
 - Preserves registers on the stack
- **Epilogue Code**
 - Restores registers saved on the stack
 - Frees stack space used for local variables
 - Executes RET or RET nn to return to caller

Prologue and epilogue code is generated automatically by a compiler in order to preserve registers, set up a stack frame, and maintain a stack after a function call completes. It is architecture and compiler specific, and understanding how this code works is critical to understanding how to debug. The following example examines prologue and epilogue code is generated for a function using the STDCALL calling convention. Most of the source code for the function has been removed for clarity.

Example: Prologue Code:

push	ebp	Saves EBP on the stack
mov	ebp, esp	Saves current value of ESP in EBP
sub	esp, 8	Subtracts 8 bytes from ESP to make room for local variables

Epilogue Code:

mov	esp, ebp	Restores ESP from EBP. This automatically cleans up space used for local variables.
pop	ebp	Restores the value of EBP.
ret	0xC	Adds 12 bytes to ESP to clean up pushed parameters from the stack before returning.

The above code is a basic prologue-epilogue sequence. Some functions need to preserve the values of registers, so that the registers can be used during the course of the function and restored to their original values before the function returns. The stack is a good place to hold the values of these registers temporarily and the prologue code will contain additional push instructions that save the value of the registers in stack memory.

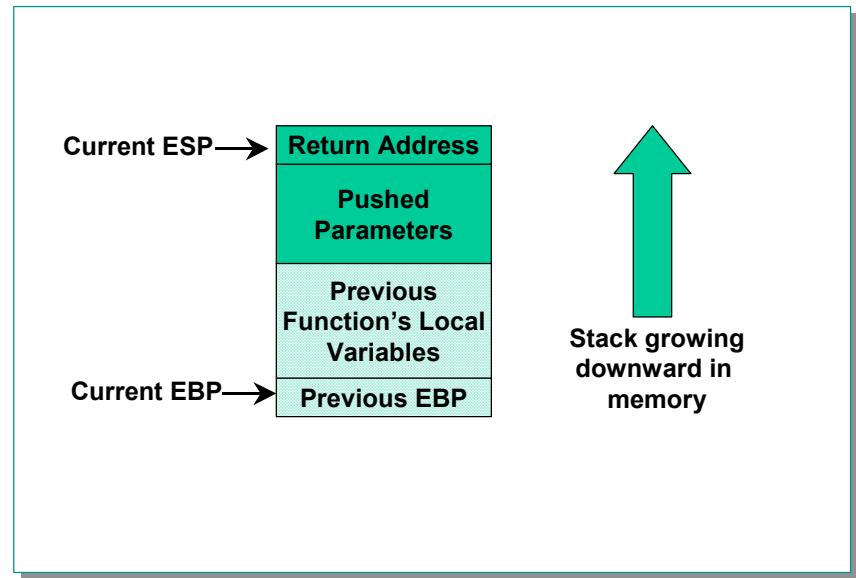
Typical Prologue Code

PUSH EBP	Save the old frame pointer on the stack.
MOV EBP, ESP	Setup the new frame pointer for this function
SUB ESP, 10	Reserve room on the stack for local variables
PUSH ESI	Preserve the value of registers used by this function
PUSH EDI	
PUSH ECX	

Here is a typical prologue code sequence. Registers are not always saved and the amount of space needed for local variables depends on the function. However this prologue code reflects the general format of a typical function's prologue.

Push	ebp	Save the old frame pointer on the stack.
mov	ebp, esp	Setup the frame pointer for this function
sub	esp, 10	Reserve room on the stack for local variables
push	esi	Preserve the value of registers used by this function
push	edi	
push	ecx	

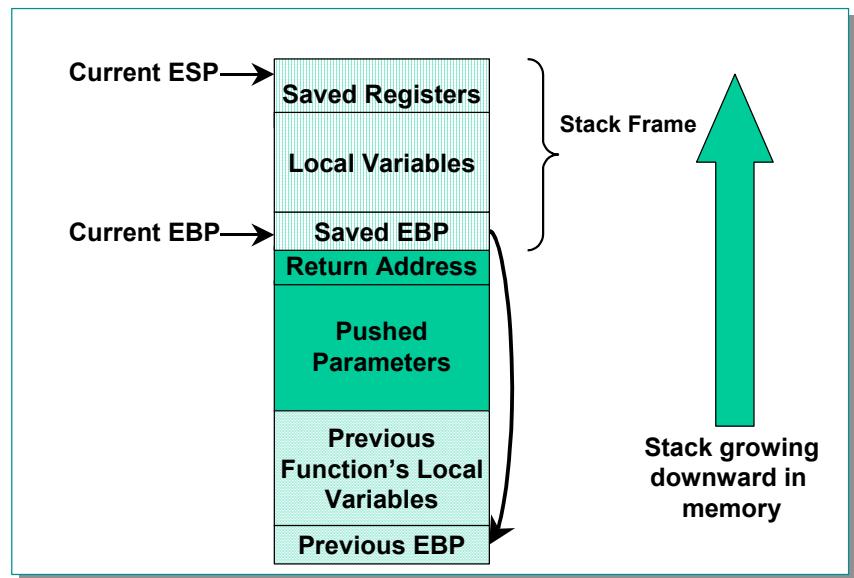
Stack Layout Before Running Prologue Code



The following is Prologue code that preserves registers:

push	ebp	Save EBP on the stack
mov	ebp, esp	Save current value of ESP in EBP
sub	esp, 8	Subtract eight bytes from ESP to make room for local variables
push push push	ebx ecx edx	Save the value of the ebx, ecx, and edx registers by pushing them onto the stack

Stack Layout After Prologue Code



The following is prologue code that preserves registers:

```

push    ebp          ; save ebp on the stack
mov     ebp, esp    ; save current value of esp in ebp
sub    esp, 8        ; subtract 8 bytes from esp to make room
                  ; for local variables
push    ebx          ; save ebx to the stack
push    ecx          ; save ecx to the stack
push    edx          ; save edx to the stack

```

Typical Epilogue Code

POP ECX	Restore the value of each register pushed in the prologue code
POP EDI	
POP ESI	
MOV ESP, EBP	Free stack space used for local variables
POP EBP	Restore the frame pointer
RET 8	Clean up pushed parameters for this function (Add 8 to ESP) .

The following is Epilogue code to restore the register values:

pop	edx	Restores the value of the ebx, ecx and edx registers.
pop	ecx	Note that the values are popped in reverse order
pop	ebx	than when pushed.
mov	esp, ebp	
pop	ebp	
ret	0xC	

Epilogue code is used to restore the state of registers before function exit. Any registers pushed in the prologue are popped in reverse order from being pushed. Some calling conventions use the RET instruction to clean up stack space used for storing the function parameters. This avoids the need for an ADD instruction to clean up the stack.

The STDCALL Calling Convention

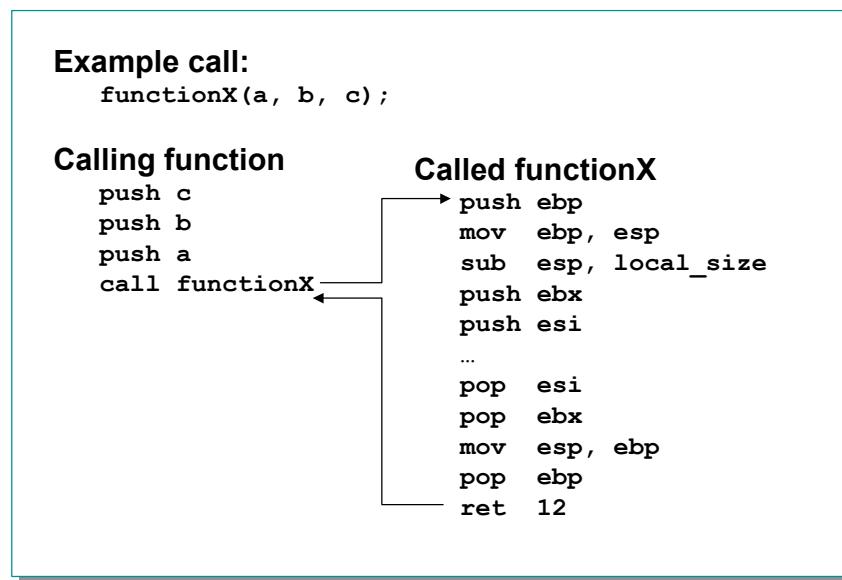
- Guidelines:
 - Arguments are passed right to left
 - Called function cleans up the stack
 - Cannot handle variable-length argument lists (vararg)
 - Default in kernel mode
- Entry point names:
 - Underscore character is prefixed to names
 - '@' sign is appended to name followed by number of bytes in the argument list
 - No case translation is performed

The STDCALL calling convention specifies that functions conform to the following guidelines:

- Arguments are passed right to left
- Called function cleans up the stack
- Underscore character is prefixed to names
- '@' sign is appended to name followed by number of bytes in the argument list
- No case translation is performed
- Cannot handle variable argument ("vararg") functions

This calling convention is used when a fixed number of arguments are required by a function. This allows the function to clean up the stack for the calling function. As a result, using the STDCALL convention can generate smaller code.

STDCALL Example



The following is the Addemup example used in earlier examples compiled using the `__stdcall` calling convention.

C Source Code:

```
#include <stdio.h>
int __stdcall Addemup(int, int);
void main(void)
{
    int x = 5;
    int y = 10;
    int z = 0;
    z = Addemup(x, y);
    printf("z= %i\n", z);
}

int __stdcall Addemup(int a, int b)
{
    int c = 0;
    c = a + b;
    return(c);
}
```

Disassembly Code:

```

addemup!main:
00401000 push    ebp          ; save base pointer
00401001 mov     ebp,esp      ; set stack pointer
00401003 sub    esp,0xc       ; make room for locals
00401006 mov     dword ptr [ebp-0x4],0x5   ; local 'x' = 5
0040100d mov     dword ptr [ebp-0x8],0xa   ; local 'y' = 10
00401014 mov     dword ptr [ebp-0xc],0x0   ; local 'z' = 0
Pushing arguments
0040101b mov     eax,[ebp-0x8]    ; load eax with local 'y'
0040101e push   eax           ; push eax onto stack
0040101f mov     ecx,[ebp-0x4]    ; load ecx with local 'x'
00401022 push   ecx           ; push ecx onto stack

00401023 call   addemup!Addemup (00401040) ; call Addemup()
00401028 mov     [ebp-0xc],eax      ; local 'z' returned in eax
0040102b mov     edx,[ebp-0xc]      ; load local 'z' into edx
0040102e push   edx           ; push edx onto stack
0040102f push   0x407030        ; push pointer to "z= %i\n"
00401034 call   addemup!printf (0040105d) ; call printf()
00401039 add    esp,0x8         ; fix stack for args
0040103c mov     esp,ebp         ; restore stack pointer
0040103e pop    ebp           ; restore base pointer
0040103f ret               ; return to caller

addemup!Addemup:
00401040 push    ebp          ; save base pointer
00401041 mov     ebp,esp      ; set stack pointer
00401043 push   ecx           ; make room for local
00401044 mov     dword ptr [ebp-0x4],0x0   ; local 'c' = 0

Arguments popped off stack
0040104b mov     eax,[ebp+0x8]    ; load eax with arg 'a'
0040104e add    eax,[ebp+0xc]    ; eax = eax + arg 'b'

00401051 mov     [ebp-0x4],eax      ; local 'c' = eax
00401054 mov     eax,[ebp-0x4]      ; load eax with local 'c'
00401057 mov     esp,ebp         ; restore stack pointer
00401059 pop    ebp           ; restore base pointer

Called routine fixes up stack
0040105a ret    0x8            ; return, clean stack

```

The CDECL Calling Convention

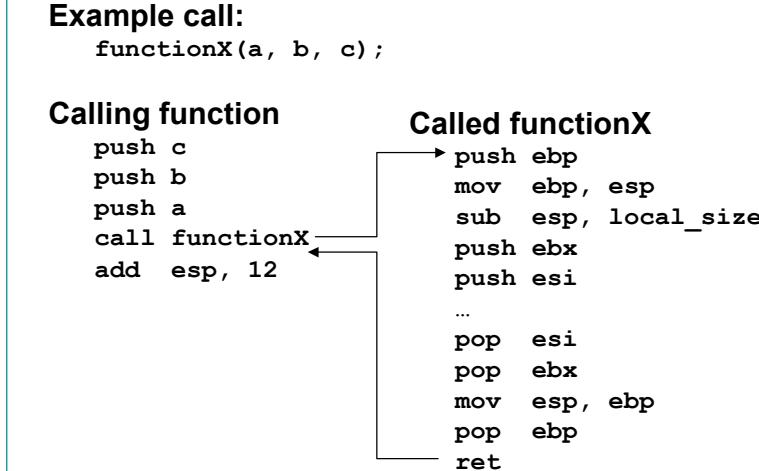
- Guidelines:
 - Default calling convention for C
 - Arguments are passed right to left
 - Calling function cleans up the stack
 - Can handle variable-length argument lists (vararg)
- Entry point names:
 - Underscore character is prefixed to names
 - No case translation is performed

The CDECL calling convention specifies that functions conform to the following guidelines.

- Default calling convention for "C"
- Arguments are passed right to left
- Calling function cleans up the stack
- Underscore character is prefixed to names
- No case translation is performed
- Can handle variable argument ("vararg") functions

This calling convention is used when a variable number of arguments are required by a function. The calling function is responsible for cleaning up the stack after the function call completes. Because the calling function cleans up the stack after the call completes and the number of arguments passed to the function is known at compile time, the compiler can generate code that allows the same function to receive a variable number of arguments.

CDECL Example



The following is the Addemup example used in earlier examples compiled using the `_cdecl` calling convention. There is no need to add this declaration to the C code since it is the compiler default.

C Source Code:

```
#include <stdio.h>
int Addemup(int, int);
void main(void)
{
    int x = 5;
    int y = 10;
    int z = 0;
    z = Addemup(x, y);
    printf("z= %i\n", z);
}

int Addemup(int a, int b)
{
    int c = 0;
    c = a + b;
    return(c);
}
```

Disassembly Code:

```

ad demup!main:
00401000 push    ebp          ; save base pointer
00401001 mov     ebp,esp      ; set stack pointer
00401003 sub    esp,0xc       ; make room for locals
00401006 mov     dword ptr [ebp-0x4],0x5   ; local 'x' = 5
0040100d mov     dword ptr [ebp-0x8],0xa   ; local 'y' = 10
00401014 mov     dword ptr [ebp-0xc],0x0   ; local 'z' = 0

Pushing arguments
0040101b mov     eax,[ebp-0x8]    ; load eax with local 'y'
0040101e push   eax           ; push local 'y' on stack
0040101f mov     ecx,[ebp-0x4]    ; load ecx with local 'x'
00401022 push   ecx           ; push local 'x' on stack

00401023 call    ad demup!Addemup    ; call Addemup

Caller fixes up the stack
00401028 add    esp,0x8        ; fix stack for args

0040102b mov     [ebp-0xc],eax    ; local 'z' returned in eax
0040102e mov     edx,[ebp-0xc]    ; load edx with local 'z'
00401031 push   edx           ; push local 'z' on the stack
00401032 push   0x407030      ; push pointer to "z= %i\n"
00401037 call    ad demup!printf  ; call printf()
0040103c add    esp,0x8        ; fix stack for args
0040103f mov     esp,ebp       ; restore stack pointer
00401041 pop    ebp           ; restore base pointer
00401042 ret             ; return

ad demup!Addemup:
00401043 push   ebp          ; save base pointer
00401044 mov     ebp,esp      ; set stack pointer
00401046 push   ecx           ; make room for local
00401047 mov     dword ptr [ebp-0x4],0x0   ; local 'c' = 0

Arguments popped off stack
0040104e mov     eax,[ebp+0x8]    ; set eax to arg 'a'
00401051 add    eax,[ebp+0xc]    ; eax = eax + arg 'b'

00401054 mov     [ebp-0x4],eax    ; local 'c' = arg'a' + arg'b'
00401057 mov     eax,[ebp-0x4]    ; set eax to local 'c'
0040105a mov     esp,ebp       ; restore stack pointer
0040105c pop    ebp           ; restore base pointer
0040105d ret             ; return to caller

```

The FASTCALL Calling Convention

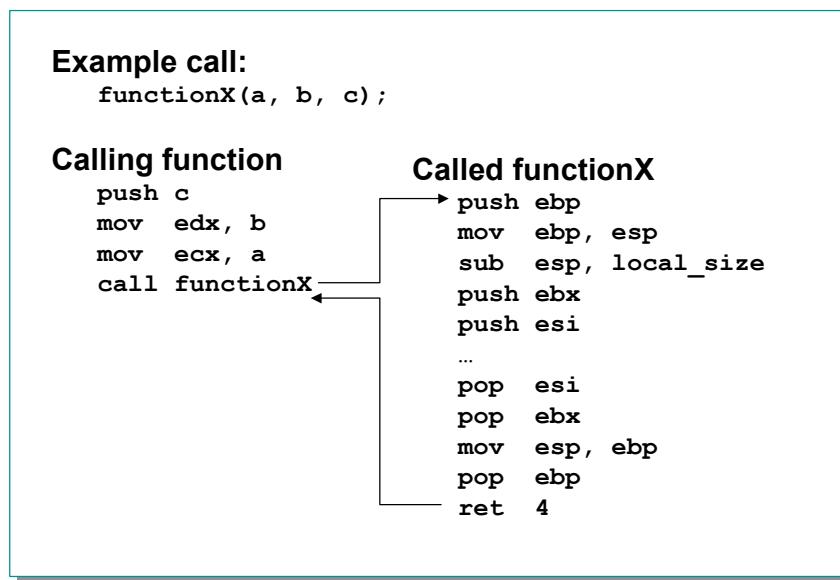
- Passes first 2 DWORD or smaller arguments in registers
- Guidelines:
 - Remaining arguments are passed right to left
 - Called function cleans up the stack (as in STDCALL)
 - Cannot handle variable length argument lists
- Entry Point Names:
 - '@' is prefixed to function name
 - '@' sign is suffixed to name followed by number of bytes in the argument list
 - No case translation is performed

The FASTCALL calling convention attempts to decrease overhead for the calling function by passing the first two DWORD or smaller arguments to the called function in registers.

fastcall calling convention follows these guidelines:

- Arguments passed right to left
- Called function cleans up the stack
- '@' is prefixed to function name
- '@' sign is suffixed to name followed by number of bytes in the argument list
- No case translation is performed
- Cannot handle variable argument ("vararg") functions

FASTCALL Example



The FASTCALL prologue and epilogue sequences handle the first two parameters to a function via registers. Therefore, these first two parameters will generally not be found on the stack. Using registers for the first two parameters allows the compiler to perform more code optimization and save the time required to push and pop parameters from stack memory.

The following is the Addemup example used in earlier examples compiled using the `__fastcall` calling convention. There is no need to add this declaration to the C code since it is the compiler default.

C Source Code:

```

#include <stdio.h>
int __fastcall Addemup(int,int);
void main(void)
{
    int x = 5;
    int y = 10;
    int z = 0;
    z = Addemup(x,y);
    printf("z= %i\n",z);
}

int __fastcall Addemup(int a, int b)
{
    int c = 0;
    c = a + b;
    return(c);
}

```

Disassembly Code:

```

addemup!main:
00401000 push    ebp          ; save base pointer
00401001 mov     ebp,esp      ; set stack pointer
00401003 sub    esp,0xc       ; make room for locals
00401006 mov    dword ptr [ebp-0x4],0x5   ; local 'x' = 5
0040100d mov    dword ptr [ebp-0x8],0xa   ; local 'y' = 10
00401014 mov    dword ptr [ebp-0xc],0x0   ; local 'z' = 0

Load arguments into registers edx and ecx
0040101b mov    edx,[ebp-0x8]    ; load local 'y' into edx
0040101e mov    ecx,[ebp-0x4]    ; load local 'x' into ecx

00401021 call   addemup!Addemup (0040103e) ; call Addemup
00401026 mov    [ebp-0xc],eax      ; local 'z' returned in eax
00401029 mov    eax,[ebp-0xc]      ; load eax with local 'z'
0040102c push   eax             ; push eax onto stack
0040102d push   0x407030        ; push pointer to "z= %i\n"
00401032 call   addemup!printf (00401061) ; call printf()
00401037 add    esp,0x8         ; fix stack for args
0040103a mov    esp,ebp         ; restore stack pointer
0040103c pop    ebp             ; restore base pointer
0040103d ret                 ; return to caller

addemup!Addemup:
0040103e push   ebp          ; save base pointer
0040103f mov    ebp,esp      ; save stack pointer
00401041 sub    esp,0xc       ; make room for locals

Load arguments from register edx and ecx
00401044 mov    [ebp-0xc],edx      ; local 'b' = edx
00401047 mov    [ebp-0x8],ecx      ; local 'a' = ecx

0040104a mov    dword ptr [ebp-0x4],0x0   ; local 'c' = 0
00401051 mov    eax,[ebp-0x8]      ; load eax with local 'a'
00401054 add    eax,[ebp-0xc]      ; eax = eax + local 'b'
00401057 mov    [ebp-0x4],eax      ; load local 'c' with result
0040105a mov    eax,[ebp-0x4]      ; load eax with local 'c'
0040105d mov    esp,ebp         ; restore stack pointer
0040105f pop    ebp             ; restore base pointer
00401060 ret                 ; return to caller

```

Note: No need to fix up stack, no arguments passed on stack

Analyzing the Stack

Interpreting the Stack

- **Consists of looking at stack and determining:**
 - **sequence of function calls**
 - **values of arguments passed to functions**
 - **values of local variables**

Based on how the stack operates, and on how the calling conventions place information onto the stack, one must be able to go to any function on the stack and determine the following in order to debug the stack:

- Sequence of function calls
- Value of passed parameters
- Value of local variables

Displaying Call Stacks

- **"K" Command Forms:**
 - **K** – Displays the basic call stack based on debugging information in the executable. It displays the frame pointer, return address, and function names.
 - **KB** – Additionally displays the first three arguments passed to each function.
 - **KV** – Additionally displays frame type-specific information and FPO data.
 - **KN** – Additionally includes frame numbers.
- **DDS range**
 - Dump Dwords as Stack

The "K" commands simplify the re-creation of the call stack by automatically reading information from the stack and displaying it as formatted text. Several forms of the command exist, each one dumping the basic plus additional information. As functions are executed and call other functions, a call stack is created in stack memory. The "K" command walks stack memory and tries to determine which functions are being executed and displays the function calls in the order that they have been executed.

"K" Command Example:

Typing 'k' dumps basic stack information including the symbolic name of the functions in the current call stack.

```
ChildEBP RetAddr
0012ff64 00401028 addemup!Addemup+0x11
0012ff80 00401143 addemup!main+0x28
0012ffc0 77e87903 addemup!mainCRTStartup+0xb4
0012fff0 00000000 KERNEL32!BaseProcessStart+0x3d
```

"KB" Command Example:

The "KB" command displays the first three DWORDS of information passed as parameters to each function on the stack.

```
ChildEBP RetAddr  Args to Child
0012ff64 00401028 00000005 0000000a 00000000 addemup!Addemup+0x11
0012ff80 00401143 00000001 00300ec0 00300e30 addemup!main+0x28
0012ffc0 77e87903 77fc9816 00270718 7ffd000 addemup!mainCRTStartup+0xb4
0012fff0 00000000 0040108f 00000000 000000c8
KERNEL32!BaseProcessStart+0x3d
```

"KV" Command Example:

The "KV" command is the same as "KB", but dumps additional information regarding FPO data.

```
ChildEBP RetAddr  Args to Child
0012ff64 00401028 00000005 0000000a 00000000 addemup!Addemup+0x11
0012ff80 00401143 00000001 00300ec0 00300e30 addemup!main+0x28
0012ffc0 77e87903 77fc9816 00270718 7ffd000 addemup!mainCRTStartup+0xb4
```

```
0012ffff0 00000000 0040108f 00000000 000000c8
KERNEL32!BaseProcessStart+0x3d (FPO: [Non-Fpo])
```

Other "K" options

The "KP" command displays the full parameter list for each function called in the stack trace. However, this requires full symbol information to work properly.

You can also add "N" to display the frame numbers (for example, "KVN").

The “DDS” Command:

The **dds** command displays double-word (4 byte) values. Each of these double-words is treated as an address in the symbol table. The corresponding symbol information is displayed for each word.

If line number information has been enabled, source file names and line numbers will be displayed if available.

Changing the Context

You can use the **.trap**, **.thread**, **.cxr**, **.exr**, and **.eexr** commands to focus the debugger on a different thread context. This allows you to display the stack of different threads with these commands.

Return Addresses (Saved EIP)

Return addresses should always equal the previous stack entry's symbolic name. In this stack,

```
ChildEBP RetAddr
0012ff64 00401028 addemup!Addemup
0012ff80 00401143 addemup!main+0x28
0012ffc0 77e87903 addemup!mainCRTStartup+0xb4
0012fff0 00000000 KERNEL32!BaseProcessStart+0x3d
```

0x00401028 should equal addemup!main+0x28

0x 0012ff80 is the ChildEBP for addemup!main+0x28

If we set a break point at Addemup!Addemup() and then allow the prologue code to run and setup the stack we can see how the base pointer and return addresses are linked to help you walk the stack..

First, let's take a look at the base pointers linked on the stack:

```
0:000> dd esp
0012ff64 0012ff80 00401028 00000005 0000000a
0012ff74 00000000 0000000a 00000005 0012ffc0
0012ff84 00401143 00000001 002f0d70 002f0db8
0012ff94 00000000 00000000 7ffd000 80100000
0012ffa4 be917d00 0012ff94 00000000 0012ffe0
0012ffb4 00402760 004070e0 00000000 0012fff0
0012ffc4 77e87903 00000000 00000000 7ffd000
0012ffd4 00000000 0012ffc8 00000000 ffffffff

0012ff64 0012ff80
0012ff80 0012ffc0
0012ffc0 0012fff0
```

Now let's take a look at the value on the stack just prior to each of the EBP's, These should be the return addresses:

```
0:000> dd esp
0012ff64 0012ff80 00401028 00000005 0000000a
0012ff74 00000000 0000000a 00000005 0012ffc0
0012ff84 00401143 00000001 002f0d70 002f0db8
0012ff94 00000000 00000000 7ffd000 80100000
0012ffa4 be917d00 0012ff94 00000000 0012ffe0
0012ffb4 00402760 004070e0 00000000 0012fff0
0012ffc4 77e87903 00000000 00000000 7ffd000
0012ffd4 00000000 0012ffc8 00000000 ffffffff

0012ff64 0012ff80 00401028
0012ff80 0012ffc0 00401143
0012ffc0 0012fff0 77e87903
```

Here we use the LN (List Nearest Symbol) debugger command to resolve the return address to actual function names:

```
0:000> ln 401028
(00401000)  addemup!main+0x28
0:000> ln 401143
(0040108f)  addemup!mainCRTStartup+0xb4
0:000> ln 77e87903
(77ea81de)  KERNEL32!BaseProcessStart+0x3d
```

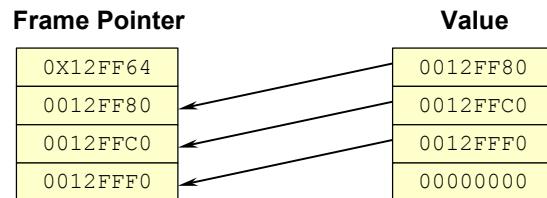
Next we use the **K** debugger command which walks the stack for you and gives you the same information:

```
0:000> k
ChildEBP RetAddr
0012ff64 00401028 addemup!Addemup
0012ff80 00401143 addemup!main+0x28
0012ffc0 77e87903 addemup!mainCRTStartup+0xb4
0012fff0 00000000 KERNEL32!BaseProcessStart+0x3d
```

This is a very important process to be able to understand and complete as the stack contains the entire history of functions that executed prior to our current position in the current thread.

Frame Pointer Linked List

- The call stack is a linked list of frame pointers
- In stack display (K), each “ChildEBP” is address of ChildEBP on next line
- FPO functions cause a break in the list



Non-FPO Pointers are a Linked List

The Frame Pointer in Non-FPO functions point to the previous functions frame pointer. Since this pattern continues, the frame pointers create a linked list.

Example:

```
0:000> !userexts.dll 12ff64
---- 0x0:
0012ff64: 0012ff80 00401028 00000005 0000000a
0012ff74: 00000000 0000000a 00000005 0012ffc0
---- 0x1:
0012ff80: 0012ffc0 00401143 00000001 002f0d70
0012ff90: 002f0db8 00000000 00000000 7ffd000
---- 0x2:
0012ffc0: 0012fff0 77e87903 00000000 00000000
0012ffd0: 7ffd000 00000000 0012ffc8 00000000
---- 0x3:
0012fff0: 00000000 00000000 0040108f 00000000
00130000: 000000c8 00000100 eeffefff 50000062
---- Total 0x4 items ----
```

The third column contains the return address of the previous function.

```
0:000> ln 401028
(00401000) addemup!main+0x28
```

The fourth and fifth columns contain the first and second parameters if applicable.

This output is similar to the kb command:

```
ChildEBP RetAddr Args to Child
0012ff64 00401028 00000005 0000000a 00000000 addemup!Addemup+0x1
0012ff80 00401143 00000001 002f0d70 002f0db8 addemup!main+0x28
0012ffc0 77e87903 00000000 00000000 7ffd000 addemup!mainCRTStartup+0xb4
0012fff0 00000000 0040108f 00000000 000000c8
KERNEL32!BaseProcessStart+0x3d
```

Using the Frame Pointer (EBP)

- Frame Pointer is used to determine the location on the stack of passed arguments and local variables
- Each ChildEBP points to previous frame's ChildEBP
- Beginning of argument list is at ChildEBP+8
- Return Address at ChildEBP+4
- First local variable at ChildEBP-4
 - (Assuming no compiler optimizations)
- Stack Pointer vs. Frame Pointer

The frame pointer is used to determine the value of passed parameters and locals for any function on the stack.

Looking at the stack from the previous examples, use the frame pointer to determine the values of the parameters passed to addemup!Addemup.

```
ChildEBP RetAddr
0012ff64 00401028 addemup!Addemup+0x1
0012ff80 00401143 addemup!main+0x28
0012ffc0 77e87903 addemup!mainCRTStartup+0xb4
0012fff0 00000000 KERNEL32!BaseProcessStart+0x3d
```

Determining the Value of Parameters

Once we know how many arguments are being passed, we can find them from the stack by simply dumping out the address of ebp+8 with the length equal to the number of variables. Example: dd 12ff64+8 L2

```
int __stdcall Addemup(int,int);
```

Here is an easy way to view it:

```
0:000> dd esp
0012ff64 0012ff80 00401028 00000005 0000000a
0012ff74 00000000 0000000a 00000005 0012ffc0
...
0012ff64 0012ff80 <= ebp of calling function
0012ff68 00401028 <= return address
0012ff6c 00000005 <= parameter 1
0012ff70 0000000a <= parameter 2

0:000> dd 12ff64+8 12
0012ff6c 00000005 0000000a
```

You should also confirm how the variables were pushed onto the stack or moved into registers by disassembling the previous function to see how it moved the data. The next

section shows the previous function prior to the call to Addemup!addemup. Note the two pushes before the call.

```
0040101b 8b45f8      mov     eax, [ebp-0x8]
0040101e 50          push    eax
0040101f 8b4dfc      mov     ecx, [ebp-0x4]
00401022 51          push    ecx
00401023 e818000000  call    addemup!Addemup (00401040)
```

Continuing with how the debugger makes stack analysis more convenient, the kb command in this case shows us the first 3 parameters being passed.

```
ChildEBP RetAddr  Args to Child
0012ff64 00401028 00000005 0000000a 00000000 addemup!Addemup+0x1
0012ff80 00401142 00000001 002f0d70 002f0db8 addemup!main+0x28
0012ffc0 77e87903 00000000 00000000 7fdf0000 addemup!mainCRTStartup+0xb4
0012fff0 00000000 0040108e 00000000 000000c8
KERNEL32!BaseProcessStart+0x3d
```

Determining the Value of Locals

Once again, looking at the slide for stdcall you can see that the locals are started one dword before the frame pointer. Thus ebp-4 is the first local variable.

It is common practice to simply unassemble the function to determine how locals are being used within the function. From the source code you can see the function has one local variable. In some cases you may see that the disassembly of the function shows that the stack is never used for locals. Instead the code is optimized to keep local variables in registers.

Frame Pointer Omission

Frame Pointer Omission (FPO)

- Optimization for x86 architecture
- Does not use the frame pointer (EBP) to save ESP contents
 - Frees EBP for use as a general purpose register
- Debugger calculates frame pointer
- Microsoft Visual C++ Compiler Options
 - /Oy - Enable Frame Pointer Omission
 - /Ox - Enable Full Optimization (Implied /Oy)
 - #pragma optimize ("y", off)

Frame Pointer Omission (FPO) is a type of optimization that suppresses or omits the creation of frame pointers on the call stack for that function. This option speeds function calls, since no frame pointers need to be setup and removed. It also frees one more register (EBP) for frequently used variables. This optimization is specific to the Intel CPU architecture.

Each of the calling conventions described thus far have saved the previous frame pointer on the stack (PUSH EBP) and has also setup the frame pointer with the current stack pointer on function entry (MOV EPB, ESP). An FPO function may save the previous frame pointer, but does not setup EBP with the current stack pointer. Instead, it uses it to hold some other variable. The debugger will calculate the frame pointer, but it must be used with caution as the usage changes based on the FPO frame type information.

This feature can be enabled in the Microsoft® Visual C++® Professional and Enterprise Editions using the /Oy compiler option.

FPO Frame Type Information

- Dump FPO information using 'kv'
- FPO data structure found in winnt.h
- (FPO: [ebp addr] [x,y,z])
 - x - Number of DWORDS pushed as parameters
 - y - Number of DWORDS allocated for local variables
 - z - Number of registers pushed in the prologue
 - ebp addr - Displayed only if EBP saved in prologue
- addemup!Addemup (FPO: [2,0,0])

FPO Data

The FPO data structure can be found in the Microsoft SDK in winnt.h and contains information that describes the contents of the stack frame. This information is used by the debugger or other programs to properly walk the stack for a FPO function. The KV commands display additional runtime information that includes FPO data. The sample stack below shows FPO data for the Addemup function in the addemup program.

```
0:000> kv
ChildEBP RetAddr
0012ff74 00401009 addemup!Addemup (FPO: [2,0,0])
0012ff80 00401115 addemup!main+0x9 (FPO: [0,0,0])
0012ffc0 77e87903 addemup!mainCRTStartup+0xb4
0012fff0 00000000 KERNEL32!BaseProcessStart+0x3d (FPO: [Non-Fpo])
```

The format of the parenthesized data is as follows:

(FPO: [ebp addr] [x,y,z])

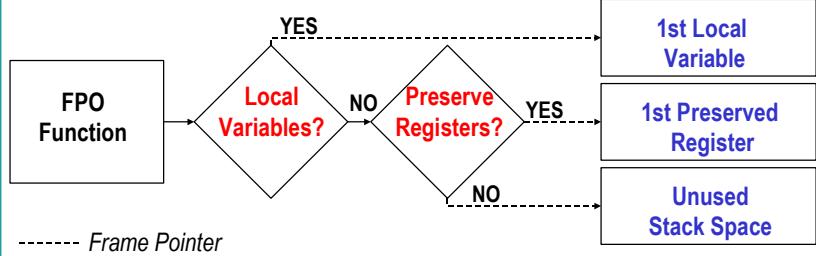
- | | |
|-----------------|--|
| x | – Number of DWORDS pushed as parameters |
| y | – Number of DWORDS allocated for local variables |
| z | – Number of registers pushed in the prologue |
| ebp addr | – Displayed only if EBP saved in prologue |

So in the case of the sample stack it can be seen that 2 DWORDS were pushed onto the stack for this function. Since the debugger had the proper symbols, it calculated a Frame Pointer for this function which is normally EBP. So FramePtr+0x8 points to the first parameter, FramePtr+0x4 points to the return address, but the FramePtr does not point to the previous EBP, since it was not saved nor setup.

Observations of FPO Functions

- Truncates Stack Without Proper Symbols
- If Function is FPO Frame Type and FASTCALL:
 - No frame pointer
 - Function parameters are not pushed onto the stack

Debuggers Calculated Frame Pointer Points to:



As long as the symbols are correct for the current and previous function, the debugger will properly calculate what the base pointer would be if the function had one.

Since EBP is simply preserved and not used to setup a stack frame, it is not necessarily the first register pushed onto the stack. This is why it no longer points to the previous EBP.

If the FPO function has any local variables, the debugger's calculated Frame Pointer points to the first local variable.

If the FPO function does not have any local variables on the stack, the debugger's calculated Frame Pointer points to the first preserved register.

If the FPO function does not have any local variables on the stack, and does not preserve any registers, the debugger's calculated Frame Pointer will point to unused stack space.

The confusion comes in when a function is of a FPO frame type and is also a FASTCALL. The function does not have a frame pointer and the parameters for the function are not pushed on the stack. However, these functions are usually small and the previous function can be disassembled to see where the registers were loaded.

Displaying a “Lost Stack”

- Debugger uses EBP to find first stack frame to begin stack backtrace
- If in an FPO procedure, EBP may not point to a stack frame
- Solution:
 - Use ESP register to find likely area of memory for stack
 - Inspect memory with DDS commands
 - Look for a likely value for EBP (four bytes earlier in memory than a saved EIP)
 - Use your values for ESP and EBP with “long form” stack backtrace command

Lost Stacks

A common result of both bugs in code, and compiler optimizations such as FPO, is that the debugger is not able to display a useful stack backtrace without help.

For example:

```
0: kd> kv
*** Stack trace for last set context - .thread/.cxr resets it
ChildEBP RetAddr  Args to Child
WARNING: Stack unwind information not available. Following frames may be
wrong.
00000000 00000000 00000000 00000000 00000000 vicamusb+0x173b
```

The reason for such a stack trace is evident if we examine the current register context: 0:

```
kd> r
eax=00000000 ebx=802ac6c8 ecx=802ac610 edx=00000000 esi=802ac6c8
edi=802ac610
eip=f07a973b esp=f0853b6c ebp=00000000 iopl=0 nv up ei pl zr na po
nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000
efl=00210246
vicamusb+173b:
f07a973b 39450c         cmp     [ebp+0xc],eax
```

ss:0010:0000000c=?The EBP register contains zero. Since the EBP register is used by the debugger as a starting point for the stack trace, this results in a useless stack display. To display a more useful stack trace, we can begin with the ESP register. The ESP register normally points to the last thing pushed on the stack, and since instructions like CALL, RET, PUSH, and POP, not to mention the processor’s trap handling logic, always use the ESP register, it is used for other purposes only in extremely unusual circumstances.

To explore the stack via the ESP register we can use the debugger’s **DDS**, Display DWORDs and Symbols, command, using the stack pointer register itself as the operand:

```
0: kd> dds esp L 60
f0853b6c  802ac610
f0853b70  802ac6c8
f0853b74  f0853bc4
f0853b78  843b3488
f0853b7c  00000000
f0853b80  00000000
```

```

f0853b84  f07a87e4 vicamusb+0x7e4
f0853b88  802ac610
f0853b8c  802ac610
f0853b90  843b3488
f0853b94  843b3488
f0853b98  00000000
f0853b9c  f0853dcc
f0853ba0  00000190
f0853ba4  802ac610
f0853ba8  843b3488
f0853bac  843b3488
f0853bb0  f0853c04
f0853bb4  00000000
f0853bb8  802ac6c8
f0853bbc  00000000
f0853bc0  8139faf0
f0853bc4  f0853c04
f0853bc8  8041d637 nt!IopfCallDriver+0x35
f0853bcc  802ac610
f0853bd0  843b3564
f0853bd4  802ac610
f0853bd8  f0853c48
f0853bdc  804bb505 nt!IopSynchronousCall+0xca
f0853be0  8139faf0
f0853be4  8139faf0
f0853be8  00000002
f0853bec  00040001
f0853bf0  00000000
f0853bf4  f0853bf4
f0853bf8  f0853bf4
f0853bfc  c00000bb
f0853c00  00000000
f0853c04  f0853c4c
f0853c08  804bb72e nt!IopRemoveDevice+0x86
f0853c0c  802ac610
f0853c10  f0853c24

```

This does seem to contain a series of saved return addresses. To provide a more useful stack trace in the familiar format, we need to supply a value for EBP to the **K** (stack backtrace) command.

Recall that in a procedure using a non-FPO stack frame, the EBP register always points to the DWORD in memory four bytes earlier than the saved EIP (instruction pointer). So, for example, f0853b80 in this example might have been the contents of EBP, just after the procedure that will return to vicamusb+0x7e4 was called.

We try this value on the **KV** command:

```

0: kd> kv =f0853b80 20
ChildEBP RetAddr  Args to Child
WARNING: Stack unwind information not available. Following frames may be
wrong.
f0853b80  f07a87e4  802ac610  802ac610  843b3488  vicamusb+0x173b
00000000  00000000  00000000  00000000  00000000  vicamusb+0x7e4

```

That doesn't look much more helpful, alas. Inspection of the **DDS** output above shows why: The saved EBP value at f0853b80 was in fact zero.

We look farther down the stack, looking for a call frame that seems to have a valid saved EBP in it, and we find a good candidate at f0853bc4. That is four bytes earlier in memory than a saved EIP value, and it seems contain an address that is in turn four bytes earlier in memory than *another* saved EIP value, and so on. This looks like it could be a chain of saved EBP values.

We test this with the **KV** command:

```

0: kd> kv =f0853bc4 20
ChildEBP RetAddr  Args to Child
WARNING: Stack unwind information not available. Following frames may be
wrong.
f0853bc4  8041d637  802ac610  843b3564  802ac610  vicamusb+0x173b
f0853bd8  804bb505  8139faf0  8139faf0  00000002  nt!IopfCallDriver+0x35 (FPO:
[0,0,2])
f0853c04  804bb72e  802ac610  f0853c24  f0853c54  nt!IopSynchronousCall+0xca

```

```
(FPO: [Non-Fpo])
f0853c4c 804bf10c 8139faf0 00000002 e239c408 nt!IopRemoveDevice+0x86 (FPO:
(Non-Fpo])
f0853c74 804bf316 802ac5f8 e214f9e8 e239c408
nt!IopDeleteLockedDeviceNode+0x24c (FPO: [Non-Fpo])
f0853cb0 80508fb0 8139faf0 e239c400 00000002
nt!IopDeleteLockedDeviceNodes+0xb0 (FPO: [Non-Fpo])
f0853d3c 805092c1 00000000 80065554 e2380848
nt!PiProcessQueryRemoveAndEject+0x7a4 (FPO: [Non-Fpo])
f0853d54 8050808b f0853d74 87d57fc8 8047333c
nt!PiProcessTargetDeviceEvent+0x33 (FPO: [EBP 0xf0853d78] [1,0,4])
f0853d78 80416ca1 87d57fc8 00000000 00000000 nt!PiWalkDeviceList+0xf7
(FPO: [Non-Fpo])
f0853da8 80452614 87d57fc8 00000000 00000000 nt!ExpWorkerThread+0xaf (FPO:
[Non-Fpo])
f0853ddc 80467122 80416bf2 00000001 00000000
nt!PspSystemThreadStartup+0x54 (FPO: [Non-Fpo])
00000000 00000000 00000000 00000000 nt!KiThreadStartup+0x16
```

This looks like a reasonable stack backtrace.

(How EBP ended up containing zero is another question. It's perfectly normal for EBP to contain something other than a valid pointer to a stack frame; the whole purpose of frame pointer optimization is to allow the EBP register to be used for something other than that purpose. It turns out that in the case of the crash dump from which this example was obtained, EBP was *not* supposed to be zero at this time, and EBP being zero was in fact the immediate cause of the crash.)

Module 8 Labs



Module 9: Crash Dump Analysis II

Module 9 Overview

- Bug Checks (Blue Screens)
- Setting the Register Context
- Additional Debugger Features

Bug Checks (Blue Screens)

Bug Checks

- All operating system failures (“blue screens”, “bug checks”, “crashes”) are result of calls to:

```
KeBugCheckEx(   BugCheckCode,
                 BugCheckArgument1,
                 BugCheckArgument2,
                 BugCheckArgument3,
                 BugCheckArgument4 );
```

- **BugCheckCode** : indicates why **KeBugCheckEx** was called
 - 0x44 = MULTIPLE_IRP_COMPLETE_REQUESTS
- Semantics of bug check arguments depend on bug check code

KeBugCheckEx

All crashes are a sign that somebody has called KeBugCheckEx.

From the Windows DDK documentation:

KeBugCheckEx brings down the system in a controlled manner when the caller discovers an unrecoverable inconsistency that would corrupt the system if the caller continued to run.

```
VOID
KeBugCheckEx(
    IN ULONG   BugCheckCode,
    IN ULONG_PTR   BugCheckParameter1,
    IN ULONG_PTR   BugCheckParameter2,
    IN ULONG_PTR   BugCheckParameter3,
    IN ULONG_PTR   BugCheckParameter4
) ;
```

Parameters

BugCheckCode

Specifies a value that indicates the reason for the bug check.

BugCheckParameterX

Supply additional information, such as the address and data where a memory-corruption error occurred, depending on the value of *BugCheckCode*.

Include

wdm.h or *ntddk.h*

Comments

A bug check is a system-detected error that causes an immediate, controlled shutdown of the system. Various kernel-mode components perform run-time consistency checking. When such a component discovers an unrecoverable inconsistency, it causes a bug check to be generated.

Whenever possible, all kernel-mode components should log an error and continue to run, rather than calling **KeBugCheckEx**. For example, if a driver is unable to allocate required resources, it should log an error so that the system continues to run; it must not generate a bug check.

A driver or other kernel-mode component should call this routine only in cases of a fatal, unrecoverable error that could corrupt the system itself.

KeBugCheckEx can be useful in the early stages of developing a driver, or while it is undergoing testing. In these circumstances, the *BugCheckCode* passed to this routine should be distinct from those codes already in use by Windows or its drivers. See [Bug Check Codes](#) for a list of these codes.

However, even during driver development, this routine is of only limited utility, since it results in a complete system shutdown. A more effective debugging method is to attach a kernel debugger to the system and then use routines that send messages to the debugger or break into the debugger. See [Kernel-Mode Debugging Routines](#) for full details.

Callers of **KeBugCheckEx** can be running at any IRQL.

Commands for Analyzing Bug Checks

- **.bugcheck**
 - Displays the bug check code and arguments from current dump
- **!analyze -v**
 - shows information about the bug check code and its arguments from current crash dump
- **Debugger documentation**
 - Each bug check is described in the “Bug Checks (Blue Screens)” → “Bug Check Code Reference” section
- **!analyze -show BugCheckCode**
 - Another way to see the bug check description -- more abbreviated than the documentation.

The debugger includes a number of commands and facilities that facilitate the analysis of bug check codes and bug check arguments.

.bugcheck

Displays the bug check code and the four arguments as recorded in the current dump.

!analyze -v

Displays “verbose” information about the bug check code, using the four bug check parameters from the current dump. This command also performs a degree of “automatic” analysis of the stack backtrace, and may suggest the identity of a suspect component.

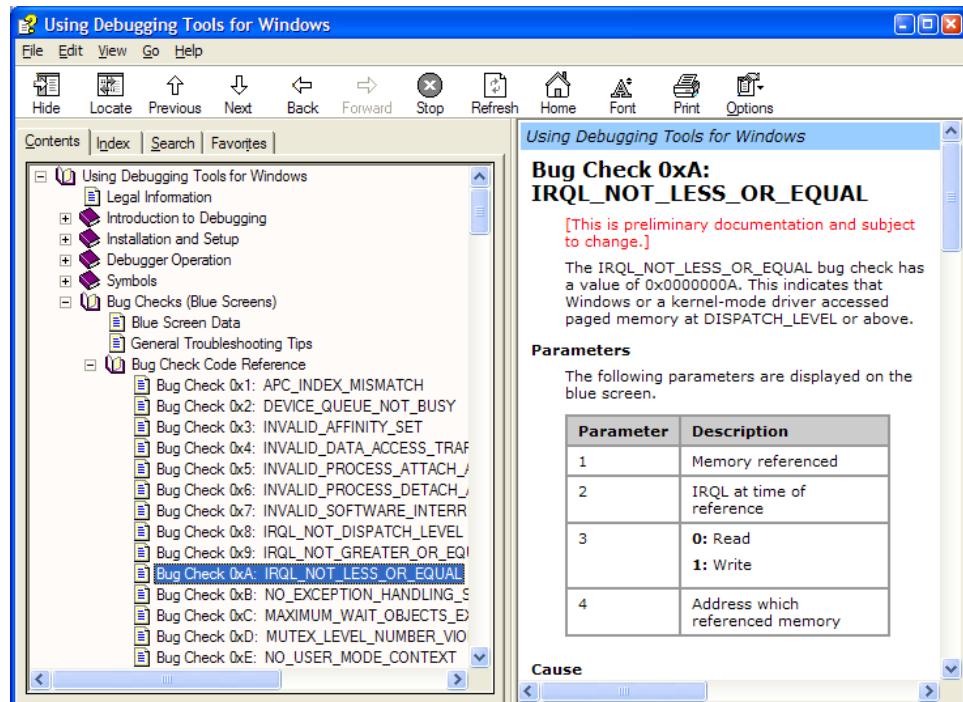
!analyze -show BugCheckCode

Displays, in the command window, a description of the bug check code and its meaning. This is usually similar, but not identical to, the corresponding information in the debugger documentation.

Debugger Documentation

The debugger documentation, in the section Bug Checks (Blue Screens) → Bug Check Code Reference, contains descriptions of almost all of the bug check codes used by the Windows operating system and its included components.

A quick reference to the standard bug check codes can also be found in the file *bugcodes.h*. This file is included in the Windows DDK. It is entirely possible, however, for bug checks to be generated using codes not found in either this file or in the debugger documentation.



Two Broad Categories of Bug Checks

- **Bug checks arising from failed assertions**
 - “In-line” code in the operating system or a driver looks for a “should never happen” condition
 - If condition detected, calls KeBugCheckEx
- **Bug checks arising from unhandled or illegal exceptions**
 - Unhandled or unhandleable exception occurs in kernel mode

All bug checks fall into one of two categories.

Bug checks arising from failed assertions are analogous to the use of the “assert” macro in C and similar languages. Code in the operating system or in device drivers sometimes performs “active” tests to check for conditions that should never happen. For example, if an attempt is made to remove an entry from a list under conditions that imply that the list should be non-empty, and the list turns out to be empty, the code might reasonably call KeBugCheckEx.

Although there are a large number of “failed assertion” bug check codes they are actually relatively few in number compared to the number of such tests that *might* be made. The operating system generally assumes that all kernel mode code is trusted, so there is little in the way of argument validation, constraint testing, and so on, in kernel mode interfaces invoked from other kernel mode interfaces. It is assumed that if mistakes are made, they will soon lead to an *unhandled or illegal exception*, and therefore to the second type of bug check described here.

The Checked Build of the operating system does include a great many more of these “assert” style tests than the Free Build. Similarly, the Driver Verifier facility provides replacements for many of the routines in the I/O Manager, replacements that do a great many more “assert” tests than the routines they replace.

The other category of bug check is the *unhandled or illegal exception*. Although *bug check codes of this category* are far less numerous than “assert” bug checks, they occur with far greater frequency.

Failed Assertion Bug Checks

- **Examples:**
 - BAD_POOL_CALLER
 - DEVICE_QUEUE_NOT_BUSY
 - MULTIPLE_IRP_COMPLETE_REQUESTS
 - NO_MORE_IRP_STACK_LOCATIONS
 - PROCESS_HAS_LOCKED_PAGES
 - Many things caught by *driver verifier* (t.b.d.)
- **Caller of KeBugCheckEx is the “reporting” routine**
 - Not necessarily that routine’s fault!

There are a very large number of bug check codes associated with failed assertions. In most cases they are extremely specific to the error condition detected.

In a failed assertion bug check, the stack trace will usually lead to a call to KeBugCheckEx. The routine that calls KeBugCheckEx is the routine that detected the problem. This is of course not a definitive indication that that routine is at fault. It may simply have detected a problem that was created by another routine – possibly farther down on the call stack, or perhaps in another thread context, or perhaps no longer executing.

Exception Bug Checks

- **Exception examples:**
 - Divide by zero
 - Page fault
 - Memory access violation
- **In user mode, exceptions can be handled by exception handlers**
 - Unhandled exception causes process termination
 - Optionally, a “just in time” debugging session can be attached to the failing process
 - Optionally, a failure report can be sent to Microsoft

Exceptions occur as a side effect of an executing instruction. They are a way for the processor to report results of instructions that are not provided for in the ordinary data paths, results that usually indicate that the processor should interrupt the normal flow of execution and do something else to cope with the problem.

For example, when an attempt is made to divide by zero, there is no possible result value that is “correct.” Division by zero has an undefined result. Furthermore there is no special result that could be returned that means “divide by zero” because all possible 32-bit integer values are possible results of valid division operations.

Although division by zero is an easily understood example of an exception, it is not a common occurrence in kernel mode. The exceptions usually associated with bug checks are related to memory access.

There are two common exceptions associated with attempts to access memory: Access violations, and page faults.

A page fault occurs when a virtual page is referenced and the page table entry describing that virtual page indicates that the page is not “valid.” Page faults, if raised in a context where they are expected, are of course handled by the operating system’s pager, and the normal flow of program execution proceeds after a brief delay.

Memory access violations occur when an attempted access to a page is denied, due to a combination of the page protection, the type of access, and the memory access mode in which the access is attempted.

Most exceptions, if raised in user mode, can be reported back to user mode and dealt with by exception handling logic in the application. (A few exceptions, like page faults, cannot be handled by anything but the operating system routines dedicated to them.) If an application does *not* handle an exception it raises, the process is normally terminated. Optionally, a “just in time” debugging session can be started.

Exception Bug Checks (Continued)

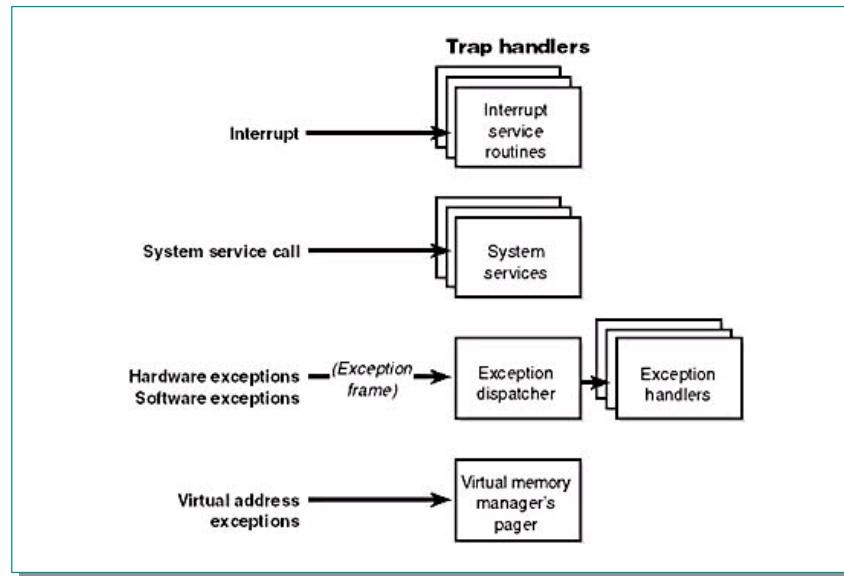
- In kernel mode, unhandled exceptions cause system crash
 - In some kernel mode contexts, exceptions can't be handled at all
- Kernel trap handler or exception dispatcher
 - Calls KeBugCheckEx
 - Can wake up debugger if kernel debugger connection already established, or if booted with /crashdebug
- Crash dump initially shows CPU state within KeBugCheckEx, not the original exception

An unhandled exception in kernel mode is always a fatal error to the operating system, resulting in a call to KeBugCheckEx. There are two ways in which the “unhandled exception in kernel mode” case can arise:

- If IRQL is 2 (DISPATCH_LEVEL) or above, exceptions are simply not permitted; it is not possible to handle them. Any exceptions raised with IRQL 2 or above are therefore “unhandled,” and result in a call to KeBugCheckEx.
- If IRQL is 0 (PASSIVE_LEVEL) or 1 (APC_LEVEL), it is possible for exception handlers to exist, be called, and to handle the exceptions raised. But if an exception is raised in kernel mode at IRQL 0 or 1, and either no exception handler is active *or* none of the active exception handlers are coded to handle this particular exception, the result is a call to KeBugCheckEx.

Analyzing crash dumps resulting from unhandled exceptions requires an additional step relative to the “assertion failure” bugchecks, that of setting the debugger’s register context. The stack trace and so on in the debugger will initially show the context within KeBugCheckEx. Fortunately there are debugger commands and procedures that allow us to reset the debugger’s register context to that of the instruction that raised the exception.

Trap Dispatching



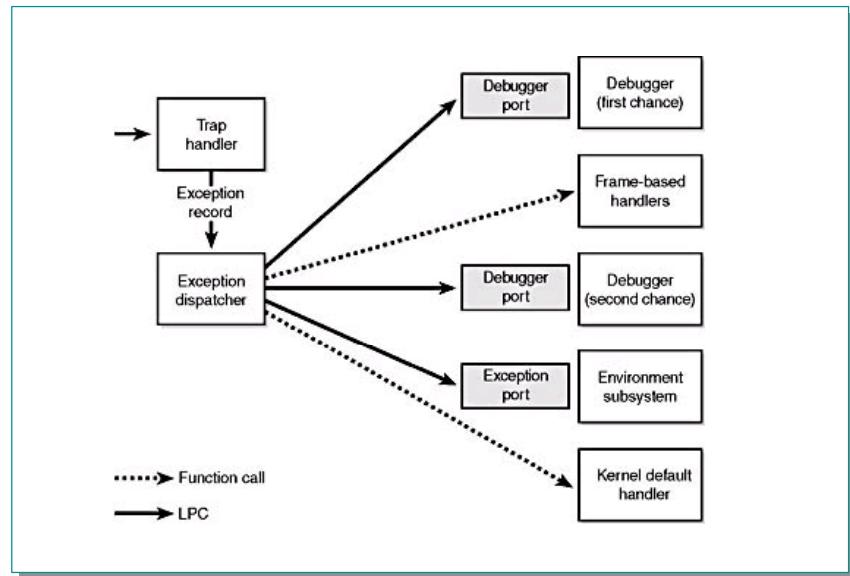
Interrupts and exceptions are conditions that divert the processor to code outside the normal flow of control. Either hardware or software can cause them. The term *trap* refers to a processor's mechanism for breaking into an executing thread when an exception or an interrupt occurs and transferring control to a fixed location in the operating system. In Windows, the processor transfers control to a *trap handler*, a function specific to a particular interrupt or exception.

The kernel distinguishes between interrupts and exceptions in the following way. An *interrupt* is an asynchronous event (one that can occur at any time) that is unrelated to what the processor is executing. Interrupts are generated primarily by I/O devices, processor clocks, or timers, and they can be enabled (turned on) or disabled (turned off). An *exception*, in contrast, is a synchronous condition that results from the execution of a particular instruction. Running a program a second time with the same data under the same conditions can reproduce exceptions. Examples of exceptions include memory access violations, certain debugger instructions, and divide-by-zero errors. The kernel also regards system service calls as exceptions (although technically they're system traps).

Either hardware or software can generate exceptions and interrupts. For example, a bus error exception is caused by a hardware problem, whereas a divide-by-zero exception is the result of a software bug. Likewise, an I/O device can generate an interrupt, or the kernel itself can issue a software interrupt (such as an APC or DPC).

When a hardware exception or interrupt is generated, the processor records enough machine state so that it can return to that point in the control flow and continue execution as if nothing had happened. To do this, the processor creates a *trap frame* on the kernel stack of the interrupted thread into which it stores the execution state of the thread. The trap frame is usually a subset of a thread's complete context. The kernel handles software interrupts either as part of hardware interrupt handling or synchronously when a thread invokes kernel functions related to the software interrupt.

Exception Dispatching



In contrast to interrupts, which can occur at any time, exceptions are conditions that result directly from the execution of the program that is running. Win32 introduced a facility known as *structured exception handling*, which allows applications to gain control when exceptions occur. The application can then fix the condition and return to the place the exception occurred, unwind the stack (thus terminating execution of the subroutine that raised the exception), or declare back to the system that the exception isn't recognized and the system should continue searching for an exception handler that might process the exception.

On the x86, all exceptions have predefined interrupt numbers that directly correspond to the entry in the IDT that points to the trap handler for a particular exception.

All exceptions, except those simple enough to be resolved by the trap handler, are serviced by a kernel module called the *exception dispatcher*. The exception dispatcher's job is to find an exception handler that can "dispose of" the exception. Examples of architecture-independent exceptions that the kernel defines include memory access violations, integer divide-by-zero, integer overflow, floating-point exceptions, and debugger breakpoints. For a complete list of architecture-independent exceptions, consult the Win32 API reference documentation.

The kernel traps and handles some of these exceptions transparently to user programs. For example, encountering a breakpoint while executing a program being debugged generates an exception, which the kernel handles by calling the debugger. The kernel handles certain other exceptions by returning an unsuccessful status code to the caller.

When an exception occurs, whether it is explicitly raised by software or implicitly raised by hardware, a chain of events begins in the kernel. The CPU transfers control to the kernel trap handler, which creates a trap frame (as it does when an interrupt occurs). The trap frame allows the system to resume where it left off if the exception is resolved. The trap handler also creates an exception record that contains the reason for the exception and other pertinent information.

Debugger breakpoints are common sources of exceptions. Therefore, the first action the exception dispatcher takes is to see whether the process that incurred the exception has an associated debugger process. If so, it sends the first-chance debug message (via an LPC port) to the debugger port associated with the process that incurred the exception. (The

message is sent to the session manager process, which then dispatches it to the appropriate debugger process.)

If the process has no debugger process attached, or if the debugger doesn't handle the exception, the exception dispatcher switches into user mode and calls a routine to find a frame-based exception handler. If none is found, or if none handles the exception, the exception dispatcher switches back into kernel mode and calls the debugger again to allow the user to do more debugging. (This is called the second-chance notification.)

All Win32 threads have an exception handler declared at the top of the stack that processes unhandled exceptions. This exception handler is declared in the internal Win32 *start-of-process* or *start-of-thread* function. The start-of-process function runs when the first thread in a process begins execution. It calls the main entry point in the image. The start-of-thread function runs when a user creates additional threads. It calls the user-supplied thread start routine specified in the *CreateThread* call.

The generic code for these internal start functions is shown here:

```
void Win32StartOfProcess(
    LPTHREAD_START_ROUTINE lpStartAddr,
    LPVOID lpvThreadParm) {

    __try {

        DWORD dwThreadExitCode = lpStartAddr(lpvThreadParm);
        ExitThread(dwThreadExitCode);

    } __except(UnhandledExceptionFilter(
        GetExceptionInformation())) {

        ExitProcess(GetExceptionCode());
    }
}
```

Notice that the Win32 unhandled exception filter is called if the thread has an exception that it doesn't handle. This function looks in the registry in the *HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug* key to determine whether to run a debugger immediately or to ask the user first.

The default "debugger" on Windows is *\winnt\system32\DrWtsn32.exe* (Dr. Watson), which isn't really a debugger but rather a postmortem tool that captures the state of the application "crash" and records it in a log file (*Drwtsn32.log*) and a process crash dump file (*User.dmp*), both found by default in the *\Documents And Settings\All Users\Documents\DrWatson* folder.

The log file contains basic information such as the exception code, the name of the image that failed, a list of loaded DLLs, and a stack and instruction trace for the thread that incurred the exception.

The crash dump file contains the private pages in the process at the time of the exception. (The file doesn't include code pages from EXEs or DLLs.) This file can be opened by WinDbg, the Windows debugger that comes with the Debugging Tools for Windows package. Because the *User.dmp* file is overwritten each time a process crashes, unless you rename or copy the file after each process crash, you'll have only the latest one on your system.

If the debugger isn't running and no frame-based handlers are found, the kernel sends a message to the exception port associated with the thread's process. This exception port, if one exists, was registered by the environment subsystem that controls this thread. The exception port gives the environment subsystem, which presumably is listening at the port, the opportunity to translate the exception into an environment-specific signal or exception. Finally, if the kernel progresses this far in processing the exception and the subsystem doesn't handle the exception, the kernel executes a default exception handler that simply terminates the process whose thread caused the exception.

Changing Contexts

The Debugger's "Contexts"

- The debugger has three "contexts" that are often used during kernel-mode debugging:
 - The *process context* determines how the debugger will interpret virtual addresses.
 - The *register context* determines how the debugger will interpret registers, and also controls the results of a stack trace. (This is sometimes called the *thread context*.)
 - The *local context* determines how the debugger will interpret local variables. This is sometimes called the *scope*.

In kernel-mode debugging, there are a large number of processes, threads, and sometimes user sessions that are executing at the same time. This means that expressions such as "virtual address 0x80002000" or "the **eax** register" are ambiguous. You need to specify the *context* in which such expressions will be understood.

The debugger has several different contexts that can be set while debugging:

- The process context determines how the debugger will interpret virtual addresses.
- The register context determines how the debugger will interpret registers, and also controls the results of a stack trace. This is sometimes called the thread context, although that term is not completely accurate.
- The local context determines how the debugger will interpret local variables. This is sometimes called the scope.

Process Context

Each process has its own page directory that records how virtual addresses are mapped to physical addresses. When any thread within a process is executing, Windows® uses this page directory to interpret virtual addresses.

During user-mode debugging, the process context is determined by the current process. Virtual addresses used in debugger commands, extensions, and debugging information windows will be interpreted according to the page directory of the current process.

During kernel-mode debugging, the process context can be set with the **.process (Set Process Context)** command. Use this command to choose which process' page directory will be used to interpret virtual addresses.

Once the process context has been set, it can be used in any commands that take addresses. Breakpoints can even be set at this address. This allows the kernel debugger to set breakpoints in user space.

The *user-mode address context* is part of the process context. Normally, you do not need to set the user-mode address context directly — setting the process context will automatically change the user-mode address context to the directory base of the relevant page table for the process. However, on an Intel Itanium processor, a single process may have more than one page directory. In this case, it may be useful to change the user-mode address context directly. This is done with the **.context (Set User-Mode Address Context)** command.

When the process context is set during kernel-mode debugging, it will keep this setting until changed by another **.process** command. The user-mode address context will keep this setting until changed by a **.process** or **.context** command. These contexts are not changed when the target computer executes, nor are they affected by changes to the register context or the local context.

Register Context

Each thread has its own register values. These values are stored in the CPU registers when the thread is executing, and are stored in memory when another thread is executing.

During user-mode debugging, the register context is usually determined by the current thread. Any reference to registers in debugger commands, extensions, and debugging information windows, will be interpreted according to the current thread's registers.

The register context can be changed to a value *other* than the current thread while performing user-mode debugging. This can be done by using the following commands:

.cxr (Display Context Record)

.ecxr (Display Exception Context Record)

During kernel-mode debugging, the register context can be controlled by a variety of debugger commands. These include:

.thread (Set Register Context)

.cxr (Display Context Record)

.trap (Display Trap Frame)

These commands do not actually cause the values of all registers to change: only the most important registers are revealed. See each command's reference page for details on how these commands work, and how they differ.

Stack traces depend on the register context. This is because the stack trace begins at the location pointed to by the stack pointer register (**esp** on an x86 processor, or **sp** on an Itanium processor). If the register context is set to an invalid or inaccessible value, stack traces cannot be obtained.

Changing the register context can also result in a change to the local context. In this way, the register context can affect the display of local variables.

There is one major exception to these rules. When debugging multiprocessor machines, some commands allow you to specify a processor. If you specify a processor for a command, the current register context is *not* used; rather, the register context matching the active thread on that processor is used. This is true even if the specified processor is the currently-active processor.

If any program execution, stepping, or tracing occurs, the register context is immediately reset to match the program counter's position. In user mode, the register context will also be reset if the current process or thread is changed.

Local Context

When a program is executing, the meaning of local variables depends on the location of the program counter, since the scope of such variables extends only to the function in which they are defined.

When performing either user-mode or kernel-mode debugging, the debugger uses the scope of the current function (the current frame on the stack) as the local context. To change this, use the **.frame (Set Local Context)** command, or double-click on the desired frame in the Calls window.

In user-mode debugging, the local context is always a frame within the stack trace of the current thread. In kernel-mode debugging, the local context is always a frame within the stack trace of the current register context's thread.

Only one stack frame at a time can be used for the local context. Local variables in other frames cannot be accessed.

The local context will be reset if any of the following occur:

- Any program execution, stepping or tracing
- Any use of the thread delimiter (~) in any command
- Any change to the register context

Setting the Register Context

- The exception causes a *trap frame* to be pushed onto the stack
 - Records CPU's registers at time of exception
 - For some exceptions in some contexts, this is converted to a *context record*
 - Rarely, this information is in a *task state segment*
- Debugger commands restore registers to those recorded at time of exception
 - .trap *TrapFrameAddress*
 - .cxr *ContextRecordAddress*
 - .tss *SegmentNumber*

To understand the procedures for analyzing crashes caused by exceptions, it is necessary to understand a little bit about how the system deals with exceptions.

When an exception (or any other trap) occurs, the processor and the kernel trap handler create a “trap frame” on the stack. This is a small data structure that describes the CPU’s register state at the time of the exception. Like a call frame in procedure calling conventions, the trap frame contains the information the trap handler needs to resume execution “in line,” after the instruction that raised the exception – if that’s possible and appropriate.

The kernel trap handler checks the context in which the exception was raised. If it’s a context (such as IRQL 2 or above) in which exceptions are not permitted at all, the kernel trap handler calls KeBugCheckEx, often with bug check code 0xA, IRQL_NOT_LESS_OR_EQUAL.

If it is possible for the exception to be handled, the kernel trap handler converts the trap frame into a *context record* that exception handlers can use. It then goes looking on the stack for declared exception handlers, and calls any that it finds in turn, until either one of them reports that it’s handled the exception, or there are no more exception handlers on the stack.

In the latter case the kernel trap handler calls KeBugCheckEx, often with bug check code 0x1E, KMODE_EXCEPTION_NOT_HANDLED.

The important point for our purposes is that the debugger can use the trap frame or the context record (or, rarely, a structure called a *task state segment*) to set the debugger’s register context to that at the time of the exception.

Setting the Register Context (continued)

- Use **!analyze -v**
 - Inspect output for STACK_COMMAND - for example:
`STACK_COMMAND: .cxr ffffffff9ef3358 ; kb`
 - Copy the command into the command window
`kd> .cxr ffffffff9ef3358 ; kb`
- Or... look on stack for:
 - A call to **PspUnhandledExceptionInSystemThread**
 - the first argument points to two DWORDs
 - first DWORD is pointer to exception record
 - second DWORD is pointer to context record
 - use these values with the **.exr** and **.cxr** commands

There are several different ways to learn how to set the debugger's register context for a particular crash dump:

- The **!analyze -v** command knows how to find trap frames or context records on the stack and to set the debugger's register context, where necessary. After executing this command, look in the output for something like

```
STACK_COMMAND: .cxr ffffffff08536fc ; kb
```

The part before the semicolon (in this case **.cxr ffffffff08536fc**) is the command to set the register context. Copy this command into the command window, and then execute any stack command, such as **kb** or **kv**.

- The debugger's documentation for each bug check code will, for most bug check codes, describe and illustrate the steps required to set the debugger's register context,
- You can follow the "heuristic" procedures on these pages.

Setting the Register Context (continued)

- If there is no call to `PspUnhandledExceptionInSystemThread`, look on the stack for one of the following:
 - A call to `KiDispatchException`
 - the third argument is pointer to trap frame
 - use that value with the `.trap` command
 - A trap frame (e.g., “TrapFrame @ xxxxxxxx”)
 - use the trap frame value with the `.trap` command
 - A task gate (e.g., “TaskGate 28:0”)
 - use the `.tss` command on the first number (e.g., `.tss 28`)
 - these are rare!
- The `.thread` command restores debugger register context to its natural value

After setting the debugger’s register context with `.trap`, `.cxr`, or `.tss`, the `.thread` command will restore it to the state it was in when you first opened the crash dump.

Additional Debugger Features

- **Workspaces**
- **Log files**
- **Source vs. disassembly mode**
- **Debugger extensions**

Types of Workspaces

- **Base Workspace**
- **Default User-Mode Workspace**
- **Remote Default Workspace**
- **Default Kernel-Mode Workspace**
- **Processor-Specific Workspace**
- **Per-process Workspace (User-Mode)**
- **Per- Dump File Workspace**

Using Workspaces

When you exit WinDbg, it saves your session configuration in a *workspace*. This allows you to easily preserve your settings from one session to another. You can also save or clear the workspaces manually, or even use a workspace to save a debugging session that is still in progress.

WinDbg has two basic types of workspaces: *default workspaces* and *named workspaces*.

Default Workspaces

WinDbg has several different types of default workspaces:

- The *base workspace* is used when WinDbg is in a dormant state.
- The *default user-mode workspace* is used when attaching to a user-mode process (with the **-p** command-line option or by using the File | Attach to a Process menu command).
- The *remote default workspace* is used when connecting to a Debugging Server.
- The *default kernel-mode workspace* is used when WinDbg begins a kernel-mode debugging session.
- The *processor-specific workspace* is used during kernel-mode debugging once WinDbg attaches to the target computer. There are separate processor-specific workspaces for x86, Intel Itanium, and Advanced Micro Devices AMD x86-64 processors.
- When WinDbg creates a user-mode process for debugging, a workspace is created for that executable. Each created executable will have its own workspace.
- When WinDbg analyzes a dump file, a workspace is created for that dump file analysis session. Each dump file will have its own workspace.

Whenever you begin a debugging session, the appropriate workspace is loaded. When you end a debugging session or exit WinDbg, a popup window appears and asks you whether you wish to save the changes you have made to the current workspace. If you start WinDbg with the **-Q** command-line option , these popup windows will not appear, and no changes will be saved.

Workspaces load in a cumulative fashion. The base workspace is always loaded first. When a particular debugging action is begun, the appropriate workspace is loaded. So most debugging will be done with two workspaces active. Kernel-mode debugging will be done with three workspaces active: the base workspace, the default kernel-mode workspace, and the processor-specific workspace.

For greatest efficiency, you should save settings in lower-level workspaces if they should be applied to all your WinDbg work.

To directly access the base workspace, start WinDbg with no target, or select **Debug | Stop Debugging** after your session is complete. You can then make any edits which are allowable in the base workspace.

There is one exception to the cumulative nature of workspaces. A workspace will not *close* a window. So if the base workspace opens the Locals window, the Locals window will remain open once debugging has begun and a higher-level workspace has become dominant.

Named Workspaces

Workspaces can also be given names and then saved or loaded individually. Once a named workspace has been loaded, all automatic loading and saving of default workspaces is disabled.

Named workspaces contain some additional information that default workspaces do not.

Opening, Saving, and Clearing Workspaces

Use the following menu commands and command-line options to control workspaces:

- The **-W** command-line option can be used to open and load a named workspace.
- The **-WF** command-line option can be used to open and load a workspace from a file.
- The **-WX** command-line option disables all automatic workspace loading. Only explicit workspace commands will cause workspaces to be saved or loaded.
- The **File | Open Workspace** menu command or the CTRL+W shortcut key can be used to open and load a named workspace.
- The **File | Save Workspace** menu command can be used to save the current default workspace or the current named workspace.
- The **File | Save Workspace As** menu command can be used to assign a name to the current workspace and save it.
- The **File | Clear Workspace** menu command allows you to delete specific items and settings from the current workspace.
- The **File | Delete Workspaces** menu command allows you to delete entire workspaces.
- The **File | Open Workspace in File** menu command can be used to open and load a workspace from a file.
- The **File | Save Workspace to File** menu command can be used to save a workspace to a file.

Information in Workspaces

- The Symbol, Executable, and Source Path
- Log file settings
- All Breakpoint Information
- All Break and Exception/Event Handling Information
- All Open Source Files
- Positions and Display Options for All Open Windows

Information in Workspaces

All default workspaces and named workspaces preserve the following information about the WinDbg graphical interface:

- The title of the WinDbg window
- The size and position of the WinDbg window on the desktop
- Which debugging information windows are open
- The size and position of each window, and the location of the Debugger Command window divider
- The default window options (Auto-Arrange, Arrange All Windows, Overlay Source, Automatically Open Disassembly)
- The default font
- Visibility of the toolbar and status bar, as well the individual toolbars on each debugging information window
- Customization of the Register window
- Flags in the Calls window and the Symbol window
- Items being viewed in the Watch window

Each workspace preserves the following information about the WinDbg configuration settings:

- The symbol path
- The executable image path
- The source path (In case of remote debugging, both the main source path and the local source path are saved.)
- Log file settings
- The COM or 1394 kernel connection settings
- The most recent paths in each file open dialog box
- The `.enable_unicode` setting

Each workspace preserves the following information about the current debugging session:

- All break and handling information for exceptions and events
- All open source files

In addition, breakpoint information is saved in workspaces, including break address and status. This means that breakpoints that are active when one session ends will be active when the next session is started. However, some of these breakpoints may be unresolved if the proper modules have not yet been loaded.

Breakpoints specified by a symbol expression, by a line number, or by using the mouse in a Source window are all saved in workspaces. To avoid ambiguity, breakpoints specified by numerical address alone, or by using the mouse in a Disassembly or Calls window, are *not* saved in workspaces.

If multiple user-mode processes are being debugged, only breakpoints associated with process zero will be saved.

Named Workspaces

Named workspaces contain additional information that is not stored in default workspaces.

This additional information includes information about the current session state. When a named workspace is saved, the current session will be saved. If this workspace is later opened, this session will be automatically restarted.

Only kernel debugging, dump file debugging, and debugging of spawned user-mode processes can be started in this way. Remote sessions and user-mode processes that the debugger attached to will not have this session information saved in their workspaces.

A named workspace of this type cannot be opened if another session is already active.

Debugging Clients and Workspaces

When WinDbg is being used as a Debugging Client, its workspace will only save values set through the graphical interface. Changes made through the Debugger Command window will not be saved. (This guarantees that only changes made by the local client will be reflected, since the Debugger Command window will accept input from all clients as well as the Debugging Server.) For more details, see Remote Debugging in the debugger documentation.

Keeping a Log File

- **Opening a New Log File**
 - **.logopen (Open Log File)**
- **Appending to an Existing Log File**
 - **.logappend (Append Log File)**
- **Closing a Log File**
 - **.logclose (Close Log File)**

The debugger can write a log file which records the debugging session. The entire contents of the Debugger Command window will be recorded in this log file: this includes the commands that you type and the responses from the debugger.

Opening a New Log File

To open a new log file, or to overwrite a previous log file, you can use the following methods:

- (*Before starting the debugger*) Set the _NT_DEBUG_LOG_FILE_OPEN environment variable.
- Use the **.logopen (Open Log File)** command.
- (*WinDbg only*) Use the **Edit | Open/Close Log File** menu command.

Appending to an Existing Log File

To append to a log file, you can use the following methods:

- (*Before starting the debugger*) Set the _NT_DEBUG_LOG_FILE_APPEND environment variable.
- Use the **.logappend (Append Log File)** command.
- (*WinDbg only*) Use the **Edit | Open/Close Log File** menu command, and check **Append**.

Closing a Log File

To close the currently-open log file, you can use the following methods:

- Use the **.logclose (Close Log File)** command.
- (*WinDbg only*) Use the **Edit | Open/Close Log File** menu command and select **Close Open Log File**.

Debugging in Assembly Mode

- **Disassembly Code**
- **Assembly Mode and Source Mode**
- **Assembly Language Source Files**

If you have C or C++ source files for your program, you can use the debugger much more powerfully if you debug in source mode.

However, there are many times you cannot perform source debugging. You may no longer have the source files for your program. You may be debugging someone else's code. You may have failed to build your executables with full .pdb symbols. And even if you can do source debugging on your application, you may need to trace Windows routines which are called by your program, or which are used to load your program.

In these cases, you will have to debug in assembly mode.

Moreover, assembly mode has many useful features that are not present in source debugging. The debugger will automatically display the contents of memory locations and registers as they are accessed, as well as displaying the address of the program counter. This makes assembly debugging a valuable tool that can be used side-by-side with source debugging.

Disassembly Code

The debugger is primarily analyzing binary executable code. Rather than displaying this in raw format, it *disassembles* this code, converting it from machine language to assembly language.

The resulting "disassembly code" can be displayed in several different ways:

- The **U (Unassemble)** command can disassemble and display a specified section of machine language.
- The **UX (Unassemble BIOS)** command can disassemble and display BIOS instructions on an x86 processor.
- (*WinDbg only*) The Disassembly window can disassemble and display a specified section of machine language. This window is automatically active if the **Window | Automatically Open Disassembly** menu command is checked. It can also be opened or made active by selecting **View | Disassembly**, pressing the ALT+7 shortcut key, or pressing the following toolbar button:



The disassembly display appears in four columns: address offset, binary code, assembly language mnemonic, and assembly language details. Here is an example of this display:

0040116b	45	inc	ebp
0040116c	fc	cld	
0040116d	8945b0	mov	eax, [ebp-0x1c]

At the right side of this line, the values of any memory locations or registers being accessed are displayed as well.

If you are viewing this in WinDbg's Disassembly window, the line representing the current program counter is highlighted in green. Lines at which breakpoints are set are highlighted as well. Enabled breakpoints are red and disabled breakpoints are yellow. If a breakpoint coincides with the current program counter, it is highlighted in purple.

Other commands are also available for manipulating assembly code:

- The **# (Search for Disassembly Pattern)** command searches a region of memory for a specific pattern. This is equivalent to searching each of the four columns of the disassembly display.
- The **A (Assemble)** command can take assembly instructions and translate them into binary machine code.

Assembly Mode and Source Mode

The debugger has two different operating modes: *assembly mode* and *source mode*.

When single-stepping through a program, the size of a single step will either be one line of assembly code or one line of source code, depending on the mode.

Several commands result in different data displays depending on the mode.

In WinDbg, the Disassembly window will automatically move to the foreground when you execute or step through a program in assembly mode. In source mode, the Source window will move to the foreground.

Setting the mode can be done in two ways:

- The **L+, L- (Set Source Options)** command can control the mode. The **L-t** command activates assembly mode.
- (*WinDbg only*) De-selecting the **Debug | Source Mode** menu item causes the debugger to enter assembly mode. This can also be done by clicking on the following toolbar button:



When WinDbg is in assembly mode is active, the **ASM** indicator on the status bar is visible.

Assembly Language Source Files

If your program was originally written in assembly language, the disassembly produced by the debugger may not precisely match your original code. In particular, NO-OPs and comments will of course not be present.

If you wish to debug your code by referencing the original *.asm* files, you need to use source mode debugging. Your assembly file can be loaded just like a C or C++ source file.

Debugging in Source Mode

- **Compilation Requirements**
- **Locating the Symbol Files and Source Files**
- **Beginning Source Debugging**
- **Source Debugging in the WinDbg GUI**
- **Source Debugging in the Debugger Command Window**
- **Source Lines and Offsets**

Debugging a program is made much easier if you can analyze the code in terms of the source, rather than the disassembled binaries.

WinDbg, NTSD, and KD can use source code in debugging, provided that the source language is C, C++, or assembly.

Compilation Requirements

For source debugging to be possible, you must build a checked (debug) version of your binaries. The compiler will create symbol files (.pdb files). These symbol files will show the debugger how the binary instructions correspond to the source lines.

The actual source files themselves must also be accessible to the debugger. The symbol files do not actually contain the text of the source files themselves.

It is best if the linker does not optimize your code. Source debugging and access to local variables are more difficult, and sometimes nearly impossible, if the code has been optimized.

Locating the Symbol Files and Source Files

To debug in source mode, the debugger needs to be able to find your source files and your symbol files.

Beginning Source Debugging

The debugger can display source information whenever it has proper symbols and source files for the thread currently being debugged.

If you launch a new user-mode application with the debugger, the initial break will occur when the application is loaded by *ntdll.dll*. Since the debugger will not have access to the *ntdll.dll* source files, you will not be able to access source information for your application at this point.

To move the program counter to the beginning of your application itself, place a breakpoint at the entry point to your binary. For example, in the Debugger Command window, type:

```
bp main
g
```

The application will then be loaded and will halt when the **main** function is entered. (Of course you can use any entry point, not just **main**.)

If the application throws an exception, it will break into the debugger. Source information will be available at this point. However, if you issue a break by using the CTRL+C, CTRL+BREAK, or **Debug | Break** commands, a new thread is created by the debugger; as a result, you will not see your source code at this point.

Once you have reached a thread for which you have source files, you can use the Debugger Command window to execute the various source debugging commands. If you are using WinDbg, the Source window will appear at this point. If you have already opened a Source window by using **Open Source File** from the **File** menu, WinDbg will usually create a *new* window for the source. The old window can be closed without affecting the debugging process.

Source Debugging in the WinDbg GUI

If you are using WinDbg, a Source window will appear as soon as the program counter is in code for which the debugger has source information.

WinDbg displays one Source window for each source file opened by you or by WinDbg itself.

You can then step through your program or execute to a breakpoint or to the cursor. If you are in source mode, the proper Source window will move to the foreground as you step through your program. Since there will be Windows routines which are called during your program's execution as well, the debugger may move a Disassembly window to the foreground when this occurs (since it will not have access to the source for these functions). When the program counter returns to known source files, the proper Source window will again become active.

As you move through your program, WinDbg will highlight your location in the Source window and the Disassembly window in green. Lines at which breakpoints are set are highlighted as well. Enabled breakpoints are red and disabled breakpoints are yellow. If a breakpoint coincides with the current program counter, it is highlighted in purple.

To activate source mode in WinDbg, use the **L+t** command, select the **Debug | Source Mode** menu item, or click on the following toolbar button:



When source mode is active, the **ASM** indicator on the status bar is grayed.

You can view or alter the values of any local variables as you step through a function in source mode.

Source Debugging in the Debugger Command Window

If you are using NTSD or CDB, you will not have the benefit of a separate Source window. However, you can still view your progress as you step through the source.

Before you can do source debugging in NTSD or CDB, you have to load source line symbols by issuing the **.lines (Toggle Source Line Support)** command, or by starting the debugger with the **-lines** command-line option.

If you execute an **L+t** command, all program stepping will be done one source line at a time. Use **L-t** to step one assembly instruction at a time. However, if you are using WinDbg, you have to select **Debug | Source Mode** or click on the source mode button to enter source mode; the **L+t** command is not sufficient.

The **L+s** command will cause the current source line and line number to be displayed at the prompt. If you only wish to see the line number, use **L+I** instead.

If you use **L+o** and **L+s**, only the source line will be displayed while you step through the program. The program counter, disassembly code, and register information will be suppressed. This allows you to quickly step through the code and view nothing but the source.

The following sequence of commands is an effective way to step through a source file:

.lines	<i>enable source line information</i>
bp main	<i>set initial breakpoint</i>
l+t	<i>stepping will be done by source line</i>
l+s	<i>source lines will be displayed at prompt</i>
g	<i>run program until "main" is entered</i>
pr	<i>execute one source line, and toggle register</i>
display off	
p	<i>execute one source line</i>

Since ENTER will repeat the last command, you can now step through the program with the enter key. Each step will cause the source line, memory offset, and assembly code to be displayed.

See Debugging in Assembly Mode for how to interpret the disassembly display.

When the assembly code is displayed, any memory location being accessed will be displayed at the right end of the line. You can use the **D*** (**Display Memory**) and **E*** (**Enter Values**) commands to view or change the values in these locations.

If you need to view each assembly instruction in order to determine offsets or memory information, use **L-t** to step by assembly instructions instead of source lines. The source line information can still be displayed. Each source line will correspond to one or more assembly instructions.

All of these commands are available in WinDbg as well as in CDB and NTSD. They can be used to view source line information from WinDbg's Debugger Command window rather than from the Source window.

Source Lines and Offsets

Another way to do source debugging is to use the expression evaluator to determine the offset corresponding to a specific source line.

The following command will output a memory offset:

```
? ` [ [module!] filename] [:linenumber] `
```

If *filename* is omitted, the debugger looks for the source file which corresponds to the current program counter.

The *linenumber* will be read as a decimal number unless preceded by an **0x**, regardless of the current default radix. If *linenumber* is omitted, the expression evaluates to the initial address of the executable corresponding to the source file.

This syntax will only be understood in NTSD or CDB if source line symbols have been loaded by the **.lines** command or the **-lines** command-line option.

This is a very versatile technique, since it can be used regardless of where the program counter is pointing. For example, it allows breakpoints to be set in advance, by using commands such as the following:

```
bp `source.c:31`
```

Using Debugger Extensions

- **Loading Debugger Extension DLLs**
 - **.load (Load Extension DLL)**
 - **.unload (Unload Extension DLL)**
 - **.unloadall (Unload All Extension DLLs)**
 - **.setdll (Set Default Extension DLL)**
 - **.chain (List Debugger Extensions)**
- **Using Debugger Extension Commands**

Loading Debugger Extension DLLs

There are several methods of loading debugger extension DLLs, as well as controlling the default debugger extension DLL and the default debugger extension path:

- (*Before starting the debugger*) Use the `_NT_DEBUGGER_EXTENSION_PATH` environment variable to set the default path for extension DLLs. This can be a number of directory paths, separated by semicolons.
- Use the **.load (Load Extension DLL)** command to load a new DLL.
- Use the **.unload (Unload Extension DLL)** command to unload a DLL.
- Use the **.unloadall (Unload All Extension DLLs)** command to unload all debugger extensions.
- Use the **.setdll (Set Default Extension DLL)** command to set the default extension DLL.

You can also load an extension DLL simply by using the full `!module.extension` syntax the first time you issue a command from that module.

The extension DLLs which you are using must match the operating system of the target machine.

The extension DLLs which ship with the Debugging Tools for Windows are each placed in a different subdirectory of the installation directory:

- The `nt4fre` directory contains extensions that should be used with the free build of Windows NT 4.0.
- The `nt4chk` directory contains extensions that should be used with the checked build of Windows NT 4.0.
- The `w2kfre` directory contains extensions that should be used with the free build of Windows 2000.
- The `w2kchk` directory contains extensions that should be used with the checked build of Windows 2000.

- The *winxp* directory contains extensions that should be used with Windows XP and Windows .NET Server.
- The *winext* directory contains extensions that can be used with any version of Windows.

If you write your own debugger extensions, you can place them in any directory. However, it is advised that you place them in a new directory and add that directory to the debugger extension path.

Using Debugger Extension Commands

The use of debugger extension commands is very similar to the use of debugger commands. The command is typed in the Debugger Command window, producing either output in this window or a change in the target application or target machine.

An actual debugger extension command is an entry point in a DLL called by the debugger.

Debugger extensions are invoked by the following syntax:

`! [module.]extension [arguments]`

The *module* name should **not** be followed with the *.dll* file name extension. If *module* includes a full path, the default string size limit is MAX_PATH characters.

If the module name is not specified, the default extension module is used. For NTSD and user-mode WinDbg, this is *ntsdexts.dll*. For KD and user-mode WinDbg, this is either *kdextalp.dll* or *kdextx86.dll* depending on the processor being debugged. The default extension module can be changed by the **.setdll** command.

If the module has not already been loaded, it will be loaded into the debugger using a call to **LoadLibrary(module)**. After the debugger has loaded the extension library, it calls the **GetProcAddress** function to locate the extension name in the extension module. The extension name is case sensitive and must be entered exactly as it appears in the extension module's .DEF file. If the extension address is found, the extension is called.

The debugger checks the user extension DLL first, and if that fails it checks the default extension DLL. If the procedure isn't found you will get an error message.

Module 9 Labs



Module 10: Kernel Debugging II

Module 10 Overview

- Live Debugging
 - Remote Debugging
 - Other Debugging Techniques
-

Live Debugging

Controlling the Target

- When the Target is Running – Stop Target With
 - WinDbg: CTRL+Break , Debug | Break, “pause” button
 - KD: CTRL+C
- When the Target is Stopped
 - Use Step or Trace commands
 - Set breakpoints
 - Use “G” (go) command, or F5, or “run” button to resume normal execution on target

While debugging a target application in user mode or target machine in kernel mode, there are two basic states. The target can be *running* or the target can be *stopped*.

When the debugger first connects to a user-mode target, it will immediately stop the target, unless the **-g** command-line option was used.

When the debugger connects to a kernel-mode target, it will leave the target running, unless the **-b** command-line option was used, or the target system has already crashed, or the target system is still halted as a result of an earlier kernel debugging action.

When the Target is Running

When the target is running, most debugger actions are unavailable.

If you wish to stop a running target, you can issue a **Break** command. This causes the debugger to *break into the target*: in other words, it will stop the target and all control will be given to the debugger. The program may not break immediately. For example, if all threads are currently executing system code, or are in a wait operation, the program will break only after control has returned to the program's code.

If a running target encounters an exception, if certain events occur, if a breakpoint is hit, or if the program terminates normally, the target will *break into the debugger*. This will stop the target and give all control to the debugger. A message will be displayed in the Debugger Command window describing the error, event, or breakpoint.

When the Target is Stopped

A variety of commands exist to start or control the target's execution:

- To cause the application to begin running, issue the **Go** command.

- To step through the program one instruction at a time, use the **Step Into** or **Step Over** commands. If a function call occurs, **Step Into** will enter the function and continue stepping through each instruction, while **Step Over** will treat the function call as a single step. When the debugger is in Assembly Mode, stepping is done one machine instruction at a time. When the debugger is in Source Mode, stepping is done one source line at a time.
- To finish the current function and stop when the return occurs, use the **Step Out** or **Trace and Watch** commands. The **Step Out** command will continue until the current function ends. **Trace and Watch** will do the same, and also display a summary of that function's calls; however, the **Trace and Watch** command has to be issued on the first instruction of the function in question.
- If an exception has occurred, the **Go with Exception Handled** and **Go with Exception Not Handled** commands can be used to resume execution and control the status of the exception.
- (*WinDbg only*) If you select a line in the Disassembly window or a Source window and then use the **Run to Cursor** command, the program will execute until this line is reached.
- (*User Mode only*) To terminate the target application and restart it from the beginning, use the **Restart** command. This command can only be used with a process which was originally created by the debugger. After the process is restarted it immediately breaks into the debugger.
- (*Kernel Mode only*) To reboot the target computer, use the **Reboot** command.
- (*WinDbg only*) To terminate the target application and clear the debugger, use the **Terminate Target** command. This allows you to begin debugging a different target.

Command Forms

Most of these commands exists in several forms. A basic text command can be used in NTSD, KD, or WinDbg. In addition, WinDbg supports command execution through menu items, toolbar buttons, and shortcut keys.

The text commands often support additional options, such as changing the location of the program counter or executing a fixed number of instructions. See the individual command pages for details.

The following table lists the forms of these commands and contains jumps to the individual reference pages:

Command Name	Text Command	WinDbg Button	WinDbg Menu	WinDbg Shortcut Key
Break	(NTSD/KD only) CTRL+C		Debug Break	CTRL+BREAK
Go	G (Go)		Debug Go	F5
Step Into	T (Trace)		Debug Step Into	F11 F8
Step Over	P (Program Step)		Debug Step Over	F10
Step Out			Debug Step Out	SHIFT+F11

Trace and Watch	W (Trace and Watch Data)			
Go with Exception Handled	GH (Go with Exception Handled)		Debug Go Handled Exception	
Go with Exception Not Handled	GN (Go with Exception Not Handled)		Debug Go Unhandled Exception	
Run to Cursor			Debug Run to Cursor	F7 CTRL+F10
Restart	.restart (Restart Target Application)		Debug Restart	CTRL+SHIFT+F5
Reboot	.reboot (Reboot Target Computer)			
Terminate Target			Debug Stop Debugging	SHIFT+F5

Using Breakpoints

- **Methods of Controlling Breakpoints**
 - BL (Breakpoint List)
 - BP (Set Breakpoint), BU (Set Unresolved Breakpoint), BA (Break on Access)
 - BC (Breakpoint Clear), BD (Breakpoint Disable), BE (Breakpoint Enabled)
- **Deferred Breakpoints**
- **Initial Breakpoints**
- **Addresses in Breakpoints**

Breakpoints are places in the executable code at which the operating system will halt execution and break into the debugger. You can then analyze the target and issue other debugger commands.

Breakpoint addresses can be specified by virtual address, by module and routine offsets, or by source file and line number (when in source mode). If the breakpoint is placed on a routine with no offset, it will be activated when that routine is entered.

There are a number of additional kinds of breakpoints. A breakpoint can be associated with a certain thread. A breakpoint can allow a fixed number of passes through an address before it is triggered. A breakpoint can automatically issue certain commands when it is triggered. And a breakpoint can be set on non-executable memory and watch for that location to be read or written to.

Methods of Controlling Breakpoints

The following methods can be used to control or display breakpoints:

- The **BL (Breakpoint List)** command will list all the current breakpoints and their current status.
- The **BP (Set Breakpoint)** command will set a new breakpoint.
- The **BU (Set Unresolved Breakpoint)** command will set a new breakpoint. Breakpoints set with **BU** are more permanent than breakpoints set with **BP**; see details below.
- The **BA (Break on Access)** command will set a *data breakpoint*. This breakpoint will be triggered if a specific memory location is accessed. (These breakpoints can be triggered when the memory location is written to, when it is read, when it is executed as code, or when kernel I/O occurs. Not all processors support all memory access breakpoints. See the command page for details.)
- The **BC (Breakpoint Clear)** command will permanently remove one or more breakpoints.
- The **BD (Breakpoint Disable)** command will temporarily disable an existing breakpoint.
- The **BE (Breakpoint Enable)** command will re-enable a disabled breakpoint.

- (*WinDbg only*) The Disassembly window and the Source windows highlight lines which have breakpoints set. Enabled breakpoints are red; disabled breakpoints are yellow; a breakpoint corresponding to the current program counter location is purple.
- (*WinDbg only*) The **Edit | Breakpoints** menu item or the ALT+F9 shortcut key will open the **Breakpoints** dialog box. This dialog box lists all breakpoints, and can be used to disable, enable, or clear existing breakpoints, or to set new breakpoints.
- (*WinDbg only*) If you set the cursor on a specific line in the Disassembly window or in a Source window and then press the following toolbar button



a breakpoint will be set at that line. If this button is pressed when the active window is neither the Disassembly window nor a Source window, this button has the same effect as **Edit | Breakpoints**.

Deferred Breakpoints

If a breakpoint is set for a routine name that has not been loaded, the breakpoint is considered *deferred*. Deferred breakpoints (also known as *unresolved* or *virtual* breakpoints) are not associated with any specific load of a module. Each time a new application is loaded, it is checked for this routine name. If this routine appears, the debugger will compute the actual coded address of the virtual breakpoints and enable it.

If a breakpoint is set by using the **BU** command, it is automatically considered unresolved. If this breakpoint is in a currently-loaded module, it is still enabled, and will function normally. However, if the module is subsequently unloaded and reloaded, this breakpoint will not vanish. (Breakpoints set by **BP** will vanish once the relevant module is unloaded.)

Initial Breakpoint

When performing user-mode debugging, an initial breakpoint automatically occurs after all statically-linked DLLs are loaded, but before any DLL initialization routines are invoked.

The **-g** command-line option will cause WinDbg or NTSD to ignore this default breakpoint. If you wish to break after DLL initialization, do not use the **-g** option. Instead, once the program has loaded and broken into the debugger, you should set a breakpoint on the **main** or **winmain** routine and then use the **G (Go)** command to proceed through initialization. The program will halt when the actual execution is about to begin.

Addresses in Breakpoints

A large variety of address syntax forms are supported. Virtual addresses, function offsets, and source line numbers are all supported. For example, any of the following commands can be used to set breakpoints:

```
bp 0040108c
bp main+5c
bp `source.c:31`
```

Reading and Writing Memory

- **Accessing Memory by Virtual Address**
 - D* (Display Memory)
 - E* (Enter Values)
- **Accessing Global Variables**
 - ?GlobalVariable = dw | dd dw l1
- **Accessing Local Variables**
 - DV (Dump Local Variables)
- **Controlling Variables Through the Watch Window**

There are several commands which allow you to access specified memory addresses or address ranges.

The following commands can read or write memory in a variety of formats. These include hexadecimal bytes, words, double words, and quad-words; short, long, and quad integers and unsigned integers; 10, 16, 32, and 64 byte real numbers, and ASCII characters.

- The **D*** (**Display Memory**) command can display the contents of a specified memory address or range.
- The **E*** (**Enter Values**) command can write a value to the specified memory address.
- (*WinDbg only*) The Memory window can display the contents of a specified memory range.

The following commands can be used to deal with more specialized data types:

- The **DT** (**Dump Type**) command can look up a great variety of data types, and display data structures which have been created by the program being debugged. This is a highly versatile command, and has many variations and options.
- The **Ds** (**Display String**) command can display a STRING or ANSI_STRING data structure.
- The **DS** (**Display String**) command can display a UNICODE_STRING data structure.
- The **DL** (**Dump Linked List**) command can trace and display a linked list.
- The **D*S** (**Display Words and Symbols**) command can look up double-words or quad-words which may contain symbol information, and then display the data and the symbol information.

The following commands can be used to manipulate entire memory ranges:

- The **M** (**Move Memory**) command can move the contents of one memory range to another.
- The **F** (**Fill Memory**) command can write a pattern to a memory range, repeating it until the range is full.

- The **C (Compare Memory)** command can compare the contents of two memory ranges.
- The **S (Search Memory)** command can search for a specified byte pattern in memory.

In most cases, these commands will interpret their parameters in the current radix. Therefore, hexadecimal addresses should be prefixed by **0x** if this radix is not 16. However, the display output of these commands is usually in hexadecimal, regardless of the current radix. (See the individual command pages for details.) The Memory window displays integers and real numbers in decimal, and displays other formats in hexadecimal. To change the default radix, use the **N (Set Number Base)** command. To quickly convert numbers from one base to another, use the **? (Evaluate Expression)** or the **.formats (Show Number Formats)** command.

Accessing Global Variables

The names of global variables are stored in the symbol files which are created when an application is compiled. The debugger interprets the name of a global variable as a virtual address. As a result, any command which accepts an address as a parameter will also accept the name of a variable as that parameter.

This means that all the commands described above can be used to read or write global variables.

In addition, you can use the **? (Evaluate Expression)** command to display the address associated with any symbol.

Here is an example. Suppose you wish to examine the global variable **MyCounter**, which is a 32-bit integer. Suppose also that the default radix is 10.

You can obtain this variable's address and then display it in the following manner:

```
0:000> ? MyCounter
Evaluate expression: 1244892 = 0012fedc
0:000> dd 0x0012fedc L1
0012fedc 00000052
```

The first command output tells you that the address of **MyCounter** is 0x0012FEDC. We then use the **D*** command to display one double-word at this address. (We could use 1244892, the decimal version of this address, instead. However, this seems unnatural to most C programmers.) The second command tells you that the value of **MyCounter** is 0x52 (decimal 82).

You could also do this in a single command, as follows:

```
0:000> dd MyCounter L1
0012fedc 00000052
```

To change the value of **MyCounter** to decimal 83, use this command:

```
0:000> ed MyCounter 83
```

Note that here we have used decimal input, since that seems more natural for an integer, and since our default radix is still 10. Note that the *output* of the **D*** command will still be in hexadecimal, however:

```
0:000> dd MyCounter L1
0012fedc 00000053
```

Accessing Local Variables

Local variables, like global variables, are stored in the symbol files. Like global variables, the debugger interprets their names as addresses. Thus they can be read and written in the same manner as global variables.

There are also additional methods of displaying and changing local variables:

- The **DV (Dump Local Variables)** command displays the names and values of all local variables.
- (*WinDbg only*) The Locals window displays the names and values of all local variables. It can also be used to change the values of these variables.

However, there is one major difference between local and global variables. When a program is executing, the meaning of local variables depends on the location of the program counter, since the scope of such variables extends only to the function in which they are defined. The debugger can use these standard scope rules, or it can override them and set a different scope.

The debugger uses the scope of the current function (the current frame on the stack) as the default scope. To change this, use the **.frame (Change Scope)** command:

```
0:000> .frame FrameNumber
```

The frame number corresponds to the position of the desired stack frame on the stack trace. To view the stack trace, use the **K (Display Stack Backtrace)** command or the Calls window. The first row has frame number zero, the second row has frame number 1, and so forth.

When the scope is changed to a new frame, the Locals window is immediately updated to reflect the new collection of local variables. The **DV** command will show the new variables as well. All these variable names will now be correctly interpreted by the memory commands described above; this allows you to read or write these variables.

Only one frame at a time can be used for the scope; local variables in other frames cannot be accessed. If any program execution, stepping, or tracing occurs, the scope is immediately reset to match the program counter's position.

When debugging optimized code, some local variables may be collapsed, replaced with a register, or perhaps even temporarily stored on the stack. If you wish to use source files or local variables while debugging, you should not optimize your code.

Controlling Variables Through the Watch Window

WinDbg has an additional way of displaying and changing global and local variables: the Watch window.

The Watch window can display any list of variables you choose. These can include global variables as well as local variables from any function. At any given time, the Watch window will display the values of those variables which match the current function's scope. These variables can have their values changed through this window as well.

Unlike the Locals window, the Watch window is not affected by the **.frame (Change Scope)** command. Only those variables which are defined in the scope of the current program counter can have their values displayed or modified.

Reading and Writing Registers and Flags

- There are two ways to manipulate CPU registers:
 - The **R (Registers)** command can be used to display or control the registers.
 - (*WinDbg only*) The Registers window can be used to display or control the registers.

Registers are small volatile storage units located on the CPU. Many of these are dedicated to specific uses, whereas others are available for use by user-mode programs.

The x86, Itanium, and AMD x86-64 processors each have a different collection of registers available.

There are two ways to manipulate CPU registers:

- The **R (Registers)** command can be used to display or control the registers. This display can be customized by using a number of options, or by using the **Rm (Register Mask)** command.
- (*WinDbg only*) The Registers window can be used to display or control the registers. It can be customized to display the registers in any order.

Registers are also automatically displayed each time the target halts. This means that if you are stepping through your code with the **P (Program Step)** or **T (Trace)** commands, you will see a register display at every step. To stop this display, use the **r** option with these commands.

On an x86 processor, the **R** command also controls a number of one-bit registers known as *flags*. Changing these flags is done by a slightly different syntax than changing regular registers.

Remote Debugging

- Remote Debugging Through the Debugger
- Remote Debugging Through *Remote.exe*

This section covers two different methods of remote debugging:

- Remote Debugging Through the Debugger
- Remote Debugging Through *Remote.exe*

Remote user-mode debugging involves two computers: the *client* and the *server*. The server is the machine with the application to be debugged.

Remote kernel-mode debugging involves three computers: the *client*, the *server*, and the *target computer*. The target computer is the machine to be debugged; the server is a machine at the same physical location as the remote host computer

Choosing the Best Method

Performing remote debugging directly through the debuggers is the more powerful of the two methods. As long as both the server and the client have the same debugger binaries installed, this is the recommended method.

The two computers do not have to be running the same version of Windows.

This is the only method which can run WinDbg remotely.

The ***Remote.exe* tool** is used to remotely control a Command Prompt window. It can be used to remotely control KD, CDB, or NTSD.

If your client does not have copies of the debugger binaries, you must use the *Remote.exe* method. This method is also required if you are running Windows 95 or Windows 98 on the client.

Remote Debugging Through the Debugger

- **Activating the Remote Debugging Server**
 - Debugger **-server tcp:port=Socket** [options]
 - Debugger **-server npipe:pipe=PipeName** {options}
- **Activating the Debugging Client**
 - Debugger **-remote tcp:server=Server,port=Socket**
 - Debugger **-remote npipe:server=Server,pipe=PipeName**
- **Running the Debugger**
- **Exiting the Debugger**

Remote debugging directly through the debugger involves running two debuggers at different locations. The debugger which is actually doing the debugging is the *Debugging Server*. The debugger which is controlling the session from a distance is the *Debugging Client*.

The two computers do not have to be running the same version of Windows.

However, the debugger binaries on the two computers must be identical versions, regardless of whether the operating systems are identical.

To set up this remote session, the *Debugging Server* is first set up, and then the *Debugging Client* is activated.

Activating the Debugging Server

There are two ways to activate the Debugging Server. It can be created when the debugger is started by using the the **-server** command-line option. It can also be created after the debugger is running by using the **.server** command.

The debuggers support two different transport protocols: TCP and NPIPE.

To start a Debugging Server from the command-line using TCP protocol, use the following syntax:

Debugger -server tcp:port=Socket [options]

To use NPIPE protocol, use this syntax:

Debugger -server npipe:pipe=PipeName [options]

After the debugger is running, the command to start a Debugging Server using TCP protocol uses the following syntax:

.server tcp:port=Socket

To use NPIPE protocol, use this syntax:

.server npipe:pipe=PipeName

The various parameters have the following possible values:

Debugger	This can be KD, NTSD, CDB, or WinDbg.
Socket	When TCP protocol is used, Socket is the socket port number.
PipeName	When NPIPE protocol is used, PipeName is a string which will serve as the name of the pipe.
options	Any additional command-line parameters can be placed here.

You can use the **.server** command to start multiple servers using different protocol options. This allows different kinds of Debugging Clients to join the session.

Activating the Debugging Client

Once the server has been activated, you can start the Debugging Client on the local computer and connect to the debugging session.

The protocol of the server must match the protocol of the client. There are two ways to start a Debugging Client: by using the **-remote** command-line option, or by using the WinDbg graphical interface.

The general syntax for starting a Debugging Client with TCP protocol is as follows:

Debugger -remote tcp:server=Server, port=Socket

The syntax for starting a Debugging Client with NPIPE protocol is as follows:

Debugger -remote npipe:server=Server, pipe=PipeName

To use the graphical interface to connect to a remote debugging session, WinDbg must be in dormant mode—that is, it must either have been started with no command-line parameters, or it must have ended the previous debugging session. Select the **File | Connect to Remote Session** menu command, or press the CTRL+R shortcut key. When the **Connect to remote debugger session** dialog box appears, enter one of the following two strings into the **Connection string** text box:

tcp:server=Server, port=Socket
npipe:server=Server, pipe=PipeName

The various parameters have the following possible values:

Debugger	This should be the same debugger as the Debugging Client.
Server	This is the network name of the machine on which the Debugging Server was created.
Socket	If TCP protocol is used, Socket is the same socket port number which was used when the server was created.
PipeName	If NPIPE protocol is used, PipeName is the name which was given to the pipe when the server was created.

Examples

Suppose you want to run WinDbg on a remote computer using TCP protocol. Your target is a failed process whose process ID is 122. The remote computer's network name is BOX17, and your socket port number is 3.

You would start the Debugging Server with the following command:

```
windbg -server tcp:port=3 -p 122
```

On the local computer, the Debugging Client would then be created in this way:

```
windbg -remote tcp:server=BOX17,port=3
```

Now suppose you want to run KD on a remote host computer using NPIPE protocol. You wish to use the **-v** command-line option. The remote computer's network name is BOX17. You decide to name the pipe "MyPipe".

You would start the Debugging Server with the following command:

```
kd -server npipe:pipe=MyPipe -v
```

On the local host computer, the Debugging Client would then be created in this way:

```
kd -remote npipe:server=BOX17,pipe=MyPipe
```

Running the Debugger

Once the remote session has been started, commands can be entered into the Debugging Server or the Debugging Client. If there are multiple clients, any of them can enter commands. Once ENTER is pressed, the command is transmitted to the Debugging Server and executed.

One exception to this rule is the setting of the source path. The Debugging Server uses the source path; each Debugging Client has its own local source path.

When a command has been executed, the debugger returns the prompt to whichever terminal entered the command. At this point, any terminal can enter a new command. If any terminal enters a command before the current command has completed, the new entry is queued; commands will be executed in the order received.

Use the **.clients** (**List Debugging Clients**) command to reveal all clients currently connected to the debugging session.

Exiting the Debugger

To terminate the Debugging Client without terminating the server, use the CTRL+B key in KD, NTSD, or CDB. In WinDbg, select **Exit** from the **File** menu.

To terminate the Debugging Server, use the **Q (Quit Debugger)** command from either the client or the server.

Remote Debugging Through *Remote.exe*

- The *Remote.exe* Utility
- Starting a *Remote.exe* Session
- Starting the *Remote.exe* Client
- *Remote.exe* Batch Files

Remote debugging through *Remote.exe* involves running the debugger on the remote computer and running the *Remote.exe* tool on the local computer.

The remote computer must be running Windows 2000 (or higher) for the debuggers to function properly. The local computer can be running any Windows operating system.

Note Since *Remote.exe* only works for console applications, it cannot be used to remotely control WinDbg.

The *Remote.exe* Utility

The *Remote.exe* utility is a versatile server/client tool which allows you to run command-line programs on remote computers.

Remote.exe provides remote network access by means of named pipes to applications that use STDIN and STDOUT for input and output. Users at other computers on a network or connected by a direct dial modem connection can either view the remote session or enter commands themselves.

This utility has a large number of uses. For example, when you are developing software, you can compile code with the processor and resources of a remote computer while you perform other tasks on your computer. You can also use *Remote.exe* to distribute the processing requirements for a particular task across several computers.

Please note that *Remote.exe* does no security authorization, and will permit anyone running *Remote.exe* Client to connect to your *Remote.exe* Server. This leaves the account under which the *Remote.exe* Server was run open to anyone who connects.

Starting a *Remote.exe* Session

There are two ways to start a *Remote.exe* session with KD or CDB. Only the second of these methods works with NTSD.

Customizing Your Command Prompt Window

The *Remote.exe* Client and *Remote.exe* Server run in Command Prompt windows.

To prepare for the remote session, you should customize this window to increase its usability. Open a Command Prompt window. Right click on the title bar and select **Properties**. Select the **Layout** tab. Go to the section titled "Screen Buffer Size" and type **90** in the **Width** box and a value between **4000** and **9999** in the **Height** box. This enables scroll bars in the remote session on the kernel debugger.

Change the values for the height and width of the "Windows Size" section if you want to alter the shape of the command prompt. Select the **Options** tab. Enable the **Edit Options** quickedit mode and insert mode. This allows you to cut and paste information in the command prompt session. Click **OK** to apply the changes. Select the option to apply the changes to all future sessions when prompted.

Starting the Remote.exe Server: First Method

The general syntax for starting a Remote.exe Server is as follows:

```
remote /s "Command_Line" Unique_Id [/f Foreground_Color] [/b Background_Color]
```

This can be used to start KD or CDB on the remote computer, as in the following examples:

```
remote /s "KD [options]" MyBrokenBox  
remote /s "CDB [options]" MyBrokenApp
```

This starts the Remote.exe Server in the Command Prompt window, and starts the debugger.

You cannot use this method to start NTSD directly, because the NTSD process runs in a different window than the one in which it was invoked. You can, however, use this method to run CDB, which is identical to NTSD except that it runs in the same Command Prompt window from which it was launched.

Starting the Remote.exe Server: Second Method

There is an alternate method that can start a Remote.exe Server. This method involves first starting the debugger, and then using the **.remote (Create Remote.exe Server)** command to start the server.

Since the **.remote** command is issued after the debugger has started, this method works equally well with KD, CDB, and NTSD.

Here is an example. First, start the debugger in the normal fashion:

```
KD [options]
```

Once the debugger is running, use the **.remote** command:

```
.remote MyBrokenBox
```

This results in a KD process which is also a Remote.exe Server with an ID of "MyBrokenBox": exactly as in the first method above.

One advantage of this method is that you do not have to decide in advance if you intend to use remote debugging. If you are debugging with one of the console debuggers and then decide that you would prefer someone in a remote location take over, you can use the **.remote** command and then they can connect to your session.

Starting the Remote.exe Client

The general syntax for starting a Remote.exe Client is as follows:

```
remote /c ServerNetBIOSName Unique_ID [/l Lines_to_Get] [/f Foreground_Color] [/b Background_Color]
```

For example, if the "MyBrokenBox" session described above was started on a local host computer whose network name was "Server2", you can connect to it with the command:

```
remote /c server2 MyBrokenBox
```

Anyone on the network with appropriate permission can connect to this debug session, as long as they know your machine name and the session ID.

Issuing Commands

Commands are issued through the Remote.exe Client and are sent to the Remote.exe Server. You can enter any command into the client as if you were directly entering it into the debugger.

To exit from the *Remote.exe* session on the Remote.exe Client, enter the **@Q** command. This leaves the Remote.exe Server and the debugger still running.

To end the server session, enter the **@K** command on the Remote.exe Server.

Remote.exe Batch Files

As a more detailed example of remote debugging with *Remote.exe*, assume the following about a local host computer in a three-computer kernel debugging scenario:

- Debugging needs to take place over a null-modem cable on COM2.
- The symbol files are in the folder *c:\winnt\symbols*.
- A log file called *debug.log* is created in *c:\temp*.

The log file holds a copy of everything you see on the Debug screen during your debug session. All input from the person doing the debugging, and all output from the kernel debugger on the target system, is written to the log file.

A sample batch file for running a debugging session on the local host is:

```
set _NT_DEBUG_PORT=com2
set _NT_DEBUG_BAUD_RATE=19200
set _NT_SYMBOL_PATH=c:\winnt\symbols
set _NT_LOG_FILE_OPEN=c:\temp\debug.log
remote /s "KD -v" debug
```

Note Make sure that either this batch file runs in the same folder containing the *Remote.exe* tool, that the folder containing *Remote.exe* is in the system path, or that the path to *Remote.exe* is explicitly defined in the batch file.

After this batch file is run, anyone with a Windows computer who is networked to the local host computer can connect to the debug session by using the following command:

```
remote /c computernname debug
```

where *computernname* is the NetBIOS name of the local host computer.

Other Debugging Techniques

- Debugging a Stalled System
- Debugging Timeouts
- Debugging Memory Leaks
- Using Driver Verifier

Debugging a Stalled System

- **Symptoms**
 - The mouse pointer can be moved, but cannot affect any windows on the screen
 - The entire screen is still and the mouse pointer does not move, but paging continues between the memory and the disk
 - The screen is still and the disk is silent
- **Some useful techniques in this case include:**
 - **Finding the Failed Process**

There are times that the computer can freeze without actually initiating a bug check. This freeze can take a variety of forms:

1. The mouse pointer can be moved, but cannot affect any windows on the screen
2. The entire screen is still and the mouse pointer does not move, but paging continues between the memory and the disk
3. The screen is still and the disk is silent

If the mouse pointer moves or there is paging to the disk, this is usually due to a problem within CSRSS.

If NTSD is running on CSRSS, press F12 and dump out each thread to see if there is anything strange.

If an examination of CSRSS reveals nothing, then the problem may be with the kernel after all.

If there is no mouse movement or paging, then it is almost certainly a kernel problem.

Analyzing a kernel crash of this sort is a difficult task. Start by breaking into KD (with CTRL+C) or WinDbg (with CTRL+BREAK). You can now use the debugger commands to examine the situation.

Some useful techniques in this case include:

- Finding the Failed Process
- Debugging Pending IRPs

Finding the Failed Process

The **!process** extension will show you the currently running process.

The parts of the process dump that are most interesting are:

- The times (a high value means that process might be the culprit)

- The handle count (this is the number in parentheses after the ObjectTable in the first line)
- The thread status (many processes have multiple threads). If the current process is *Idle*, chances are the machine is either truly idle or hung due to some unusual problem.

Although using the **!process 0 7** extension is the best way to find the problem on a hung system, it is sometimes too much information to filter. It would be worthwhile to first do a **!process 0 0** and do a **!process** on the process handle for CSRSS and any other suspicious processes.

When using a **!process 0 7**, many of the threads might say "kernel stack not resident" because those stacks are paged-out. If those pages are still in the cache in transition, you can get more information by using a **.cache decodeptes** before **!process 0 7**:

```
kd> .cache decodeptes
kd> !process 0 7
```

If the guilty process can be identified, use **!process <process> 7** to show the kernel stacks for each thread in the process. This can indicate what the problem could be in kernel mode and what the suspect process is calling.

Apart from **!process**, there are several other extensions which help track down the causes for hung machines:

Extension	Effect
!ready	Identifies the threads that are ready to run in order of priority.
!locks	Identifies any held resource locks, in case it is a deadlock with retail timeouts.
!vml	Checks the virtual memory usage.
!poolused	Checks to see if any one type of pool allocation is huge (pool tagging required).
!memusage	Checks the physical memory status.
!irpfind	Searches nonpaged pool for active IRPs.

If none of these point you to anything strange, try setting a breakpoint at KiSwapThread (**bp ntoskrnl!KiSwapThread**) to see if you are stuck in one process or if you're still scheduling other processes. If you are not stuck, you could try setting some breakpoints in common functions like **NtReadFile** to see if things are getting stuck down a specific code path.

Debugging Timeouts

- Resource Timeouts (kernel)
 - !locks
 - !thread
- Critical Section Timeouts (user)
 - RtlpWaitForCriticalSection
 - !locks

There are two main timeouts which occur on Windows® systems:

- Resource timeouts (kernel mode)
- Critical section timeouts (user mode)

In many cases, these problems are simply a matter of a thread taking too long to release a resource or exit a section of code.

On a retail system, the timeout value is set high enough that you would not see the break (a true deadlock would simply hang). The timeout values are set in the registry under **HKEY_LOCAL_MACHINE\CurrentControlSet\Control\SessionManager** in seconds.

Resource Timeouts

During a resource timeout, the thread waiting for the resource will break into the kernel debugger with a message like:

```
Resource @ 800e99c0
ActiveCount = 0001 Flags = IsOwnedExclusive sharedWaiter
NumberOfExclusiveWaiters = 0000
Thread = 809cd2f0, Count = 01
Thread = 809ebc50, Count = 01
Thread = 00000000, Count = 00
Thread = 00000000, Count = 00
Thread = 00000000, Count = 00
NT!DbgBreakPoint+0x4:
800cee04: 000000ad callkd
```

The thread that holds the lock is the first thread listed (or multiple threads if it is owned shared). To examine that thread, use a **!thread** extension on the thread ID (809cd2f0, in the example above). This will give a stack for the thread owning the resource. If it is also waiting on a resource, **ExpWaitForResourceExclusive** (or **ExpWaitForResourceShared**) will be on the stack for that thread.

The first parameter to **ExpWaitForResourceXxx** is the lock which is being waited on. To find out about that resource, use a **!locks <resource id>** extension which will give you another thread to check.

If you get to a thread that is not waiting for another resource, that thread is probably the source of the problem. For a list of all held locks, use a **!locks** extension with no parameter at the **kd>** prompt.

Example

```
Resource @ fc664ee0 //here's the resource lock address

ActiveCount = 0001 Flags = IsOwnedExclusive ExclusiveWaiter
NumberOfExclusiveWaiters = 0001
Thread = ffaf5410, Count = 01 //here's the owning thread
Thread = 00000000, Count = 00
ntoskrnl!_DbgBreakPoint:
80131400 cc int 3 //start with a stack
kd> kb
ChildEBP RetAddr Args to Child
fcd44980 801154c0 fc664ee0 ffb45d0 ntoskrnl!_DbgBreakPoint
fcd4499c 80102521 fc664ee0 ffb08ea8 ntoskrnl!_ExpWaitForResource+0x114
// lock being waited on...

kd> !locks fc664ee0 //locks resource address gives
lock info
Resource @ nwrdr!_NwScavengerSpinLock (0xfc664ee0) Exclusively owned
Contention Count = 45
NumberOfExclusiveWaiters = 1
Threads: ffaf5410-01 //owning thread again
1 total locks, 1 locks currently held
kd> !thread ffaf5410 //check the owning thread

THREAD ffaf5410 Cid e7.e8 Peb: 7ffdde000 WAIT: (Executive) KernelMode
Non-Alertable
feecf698 SynchronizationEvent
IRP List:
fef29208: (0006,00b8) Flags: 00000884 Mdl: feed8328
Not impersonating
Owning Process ffaf5690
WaitTime (seconds) 2781250
Context Switch Count 183175
UserTime 0:00:23.0153
KernelTime 0:01:01.0187
Start Address 0x77f04644
Initial Sp fec6c000 Current Sp fec6b938
Priority 11 BasePriority 7 PriorityDecrement 0 DecrementCount 8

ChildEBP RetAddr Args to Child
fec6b950 801044fc feecf668 feecf668 ntoskrnl!KiSwapContext+0x25
fec6b974 fc655976 feecf698 00000000 ntoskrnl!_KeWaitForSingleObject+0x218
fec6ba5c fc6509fa e1263968 fef29208 nwrdr!_ExchangeWithWait+0x38
fec6ba28 fc6533e5 feecf668 e125b3c8 nwrdr!_CreateScb+0x2ff
fec6bac0 fc652f26 feecf668 fec6bae4 nwrdr!_CreateRemoteFile+0x2c9
fec6bb6c fc652b14 feecf668 fef29208 nwrdr!_NwCommonCreate+0x3a2
fec6bbac 80107aea fee50b60 fef29208 nwrdr!_NwFsdCreate+0x56
fec6bbc0 80142792 fef37700 fec6bdbc ntoskrnl!IoCallDriver+0x38
fec6bd10 80145403 fee50b60 00000000 ntoskrnl!_IopParseDevice+0x6a0
fec6bd7c 80144c0c 00000000 fec6be34 ntoskrnl!_ObpLookupObjectName+0x479
fec6be5c 80127803 0012dd64 00000000 ntoskrnl!_ObOpenObjectByName+0xa2
fec6bef4 801385c3 0012dd64 0012dd3c ntoskrnl!_NtQueryAttributesFile+0xc1
fec6bef4 77f716ab 0012dd64 0012dd3c ntoskrnl!_KiSystemService+0x83
0012dd20 00000000 00000000 00000000 ntdll!_ZwQueryAttributesFile+0xb
```

Critical Section Timeouts

Critical section timeouts can be identified by the stack trace that has **RtlpWaitForCriticalSection** near the top of the stack. Another variety of critical section timeout is a possible deadlock application error. To debug critical section timeouts properly, CDB or WinDbg is necessary.

Similar to resource timeouts, **!locks** extension will give a list of locks currently held and the threads which own them. Unlike resource timeouts, the thread IDs given are not

immediately useful. These are system IDs which do not map directly to the thread numbers used by CDB.

To map the system thread IDs to CDB thread numbers, use the ~ (tilde) command. This will list all threads currently being debugged and give their thread number, ID, Thread Environment Block (TEB), and state. Just as with `ExpWaitForResourceXxx`, the lock identifier is the first parameter to `RtlpWaitForCriticalSection`. Continue tracing the chain of waits until either a loop is found or the final thread is not waiting for a critical section timeout.

Example of Debugging a Critical Timeout

```
0:024> kb      start with a stack

ChildEBP RetAddr  Args to Child
0569fcfa4 77f79c78 77f71000 002a6b88 ntdll!_DbgBreakPoint
0569fd04 77f71048 5ffa9f9c 5fef0b4b ntdll!_RtlpWaitForCriticalSection+
0569fd0c 5fef0b4b 5ffa9f9c 002a6b88 ntdll!_RtlEnterCriticalSection+0x48
0569fd70 5fedf83f 002a6b88 0569fdc0 winsrv!_StreamScrollRegion+0x1f0
0569fd8c 5fedfa5b 002a6b88 00190000 winsrv!_AdjustCursorPosition+0x8e
0569fdc0 5fedf678 0569ff18 0031c200 winsrv!_DoWriteConsole+0x104
0569fefc 5fe6311b 0569ff18 0569ffd0 winsrv!_SrvWriteConsole+0x96
0569ffff4 00000000 00000000 00000024 csrsrv!_CsrApiRequestThread+0x4ff
0:024> !locks      //!locks to find the critical section

CritSec winsrv!_ScrollBufferLock at 5ffa9f9c // 5ffa9f9c is the first one
LockCount      5
RecursionCount 1
OwningThread   88           //here's the owning thread ID
EntryCount     11c
ContentionCount 135
*** Locked

CritSec winsrv!_gcsUserSrv+0 at 5ffa91b4 //second crit sect found below

LockCount      8
RecursionCount 1
OwningThread   6d           //second owning thread
EntryCount     1d6c
ContentionCount 1d47
*** Locked
0:024> ~      //tilde to list CDB threads to find thread IDs
0 id: 16.15  Teb 7fffd000 Unfrozen
1 id: 16.13  Teb 7ffdb000 Unfrozen
2 id: 16.30  Teb 7ffda000 Unfrozen
3 id: 16.2f  Teb 7ffd9000 Unfrozen
4 id: 16.2e  Teb 7ffd8000 Unfrozen
5 id: 16.6c  Teb 7ff6c000 Unfrozen
6 id: 16.6d  Teb 7ff68000 Unfrozen //second owning thread
7 id: 16.2d  Teb 7ffd7000 Unfrozen
8 id: 16.33  Teb 7ffd6000 Unfrozen
9 id: 16.42  Teb 7ff6f000 Unfrozen
10 id: 16.6f  Teb 7ff6e000 Unfrozen
11 id: 16.6e  Teb 7ffd5000 Unfrozen
12 id: 16.52  Teb 7ff6b000 Unfrozen
13 id: 16.61  Teb 7ff6a000 Unfrozen
14 id: 16.7e  Teb 7ff69000 Unfrozen
15 id: 16.43  Teb 7ff67000 Unfrozen
16 id: 16.89  Teb 7ff50000 Unfrozen
17 id: 16.95  Teb 7ff65000 Unfrozen
18 id: 16.90  Teb 7ff64000 Unfrozen
```

```

19  id: 16.71   Teb 7ff63000 Unfrozen
20  id: 16.bb   Teb 7ff62000 Unfrozen
21  id: 16.88   Teb 7ff61000 Unfrozen //first owning thread
22  id: 16.cd   Teb 7ff5e000 Unfrozen
23  id: 16.c1   Teb 7ff5f000 Unfrozen
24  id: 16.bd   Teb 7ff5d000 Unfrozen
0:024> ~21s                                // switch to owning thread
ntdll!_ZwWaitForSingleObject+0xb:
77f71bfb c20c00      ret     0xc
0:021> kb          //get a stack; it's also waiting on a crit sect

ChildEBP RetAddr  Args to Child
0556fc44 77f79c20 00000110 00000000 ntdll!_ZwWaitForSingleObject+0xb
0556fcb0 77f71048 5ffa91b4 5feb4f7e ntdll!_RtlpWaitForCriticalSection+
0556fcb8 5feb4f7e 5ffa91b4 0556fd70 ntdll!_RtlEnterCriticalSection+0x48
0556fcf4 5fef0b76 01302005 00000000 winsrv!__ScrollDC+0x14
0556fd70 5fedf83f 002bd880 0556fdc0 winsrv!_StreamScrollRegion+0x21b
0556fd8c 5fedfa5b 002bd880 00190000 winsrv!_AdjustCursorPosition+0x8e
0556fdc0 5fedf678 0556ff18 002bd70 winsrv!_DoWriteConsole+0x104
0556fefc 5fe6311b 0556ff18 0556ffd0 winsrv!_SrvWriteConsole+0x96
0556fff4 00000000 00000000 00000024 csrssrv!_CsrApiRequestThread+0x4ff
0:021> ~6s                                //scroll up to find crit sect and thread number
winsrv!_PtifFromThreadId+0xd:
5fe8429a 394858      cmp     [eax+0x58],ecx
ds:0023:7f504da8=000000f8
0:006> kb          // get the final stack
ChildEBP RetAddr  Args to Child
01ecfeb4 5fec0d7 00000086 00000000 winsrv!_PtifFromThreadId+0xd
01ecfed0 5feccf62 00000086 01ecfff4 winsrv!__GetThreadDesktop+0x12
01ecfefc 5fe6311b 01ecff18 01ecffd0 winsrv!__GetThreadDesktop+0x8b
01ecfff4 00000000 00000000 00000024 csrssrv!_CsrApiRequestThread+0x4ff

```

Debugging Memory Leaks

- **Kernel-Mode Memory Leaks**
 - **Using Performance Monitor to Detect a Leak**
 - **Using Poolmon.exe to Find Kernel-Mode Memory Leaks**
 - **Using the Kernel Debugger to Find Leaks**

Kernel Mode Memory Leaks

A kernel-mode memory leak is caused by a process that allocates memory from either paged or non-paged pools for use, but does not later free it up when done. The result is that these limited pools of memory become depleted over time, often causing the system to stop functioning properly.

Using Performance Monitor to Detect a Leak

Without doing any preparation, you can check for major memory leaks using the Performance MMC utility. We will not use Perfmon to find the source of the leak but, rather, to indicate that a leak is occurring.

Start Perfmon on the target computer. Add the following counters:

- Memory - Pool NonPaged Bytes
- Memory - PoolPaged Bytes
- PageFile - Usage %

You may want to change the update time to 600 seconds to get a good graphical picture of the leak over time. You might also want to log the data to a file for later examination.

Start the application or test that you believe is causing the leak. It might be a local application, network access, and so on. Do not use the target computer while the test is in progress.

Watch the perfmon counters over time. Leaks are usually slow and may take hours to detect. Wait for a few hours before you make any judgments. Starting your test will cause the counters to jump. It may take a while for the memory pools to reach a steady state. If a paged pool leak is occurring, both Memory - PoolPaged Bytes and PageFile - Usage % will increase over time. If a NonPaged Pool leak is occurring, only the Memory - PoolNonPage Bytes counter will increase. Leaks here are major and should be reported quickly.

Using Poolmon.exe to Find Kernel-Mode Memory Leaks

Poolmon.exe is a Windows Support Tool used to monitor pool memory usage by tags. You must modify the GlobalFlag settings to enable pool tagging before poolmon.exe will work. The easiest way to do this is to run the gflags utility and check the box next to **Enable pool tagging**.

The Poolmon.exe header allows you to monitor total Paged and Nonpaged pool bytes. The columns show usage for each tag name. It also updates the screen every few seconds. Here is a quick example:

```
Memory: 16224K Avail: 4564K PageFlts: 31 InRam Krnl: 684K P: 680K
Commit: 24140K Limit: 24952K Peak: 24932K Pool N: 744K P: 2180K
```

Tag	Type	Allocs	Frees	Diff	Bytes	Per Alloc
CM	Paged	1283	(0)	1002	(0)	281
Strg	Paged	10385	(10)	6658	(4)	3727
Fat	Paged	6662	(8)	4971	(6)	1691
MmSt	Paged	614	(0)	441	(0)	173
					83456	(0)
						482

Poolmon.exe also has a few command keys that sort the output for you. Press the letter below to perform the operation. It takes a few seconds for each command to work.

Here is a list of a few of the commands:

P - Sorts tag list by Paged, Non-Paged, or mixed – cycles between these three.

B - Sorts tags by max byte usage.

M - Sorts tags by max byte allocation.

T - Sort tags alphabetically by tag name.

E - Display totals across bottom – cycles between paged and non-paged totals.

A - Sorts tags by allocation size.

F - Sorts tags by "frees".

S - Sorts tags by the differences of allocs and frees.

Q - Quit.

Example: Say that you are looking for a Paged pool leak. Start Poolmon.exe. Press P once (wait a few seconds) and it sorts by Nonpaged. Press P again and it sorts by Paged pool. Now Press B to sort by biggest byte users. Start your test. Take a screen shot and copy it to notepad. Take another screen shot every half hour or so. You will be able to see which tag's bytes are increasing. Stop your test and wait a few hours. See how much of this tag is freed up. In the example above, the tag "Strg" was growing very quickly and never freed up as much as it allocated.

Using the Kernel Debugger to Find Leaks

You can set the debugger to break whenever a specific pool tag is used by setting the global system variable "poolhittag" to the tag you believe is the source of the memory leak. You can set this global variable by using the syntax: "ed poolhittag 'tag'" The tag is case sensitive and must be entered in reverse.

Example: ed poolhittag 'eliF'

The debugger will break every time pool is allocated or freed with the tag 'File' for file objects. This is a time consuming process and may take a while. When the debugger breaks on these allocations and frees you can use the KB debugger command to view the stack trace associated with the alloc or free. Another piece of information that can be very

helpful in tracking down leaks is the allocation size. If this is known it will be much easier to isolate the function allocating the memory.

Once you think you have determined where the memory is being allocated, you can unassemble the function and change the tag to something unique and then allow the system to run for a set period of time. Then you could break back in with the debugger and use the !poolfind kernel debugger extension to find all pool allocation with the new unique tag.

Using Driver Verifier

- **Activation: Driver Verifier Manager**
 - Start | Run | “verifier”
- **Automatic Checks – always activated**
 - Monitors IRQL and memory routine use
 - Prevents stack switching
 - Makes sure drivers clean up when they unload
- **Additional Options**
 - Can activate any or all of these options

Driver Verifier

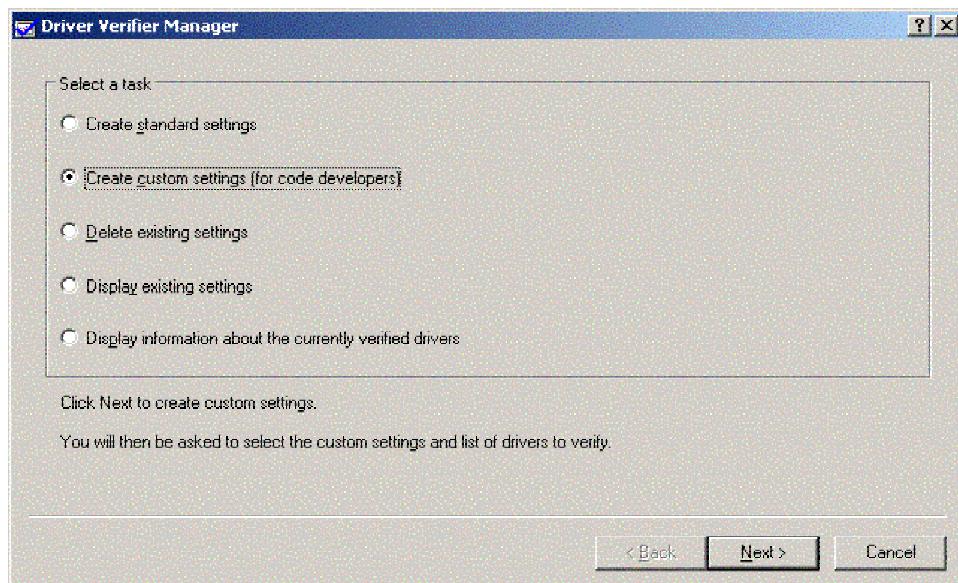
Driver Verifier is a tool that can monitor kernel-mode drivers and graphics drivers to verify that they are not making illegal function calls or causing system corruption. It can perform a large number of checks on these drivers, as well as subject them to a variety of stresses and tests to flush out improper behavior.

Driver Verifier can be used on any number of drivers simultaneously, or on one driver at a time. It has various options that can be enabled or disabled. This allows you to put a driver through heavy stresses or through a more streamlined test.

Microsoft uses Driver Verifier to check all device drivers that vendors submit for Hardware Compatibility List (HCL) testing. Doing so ensures that the drivers on the HCL are compatible with Windows and free from common driver errors.

Driver Verifier Configuration and Initialization

To configure Driver Verifier and view statistics about its operation, run the Driver Verifier Manager (**Start | Run | “verifier”**). The Driver Verifier Manager appears:



Using Driver Verifier Manager, you can choose which options you wish to activate, and which drivers you wish to verify.

Driver Verifier Options

The following options are available:

- **Automatic Checks**

These checks are always performed on a driver that is being verified, regardless of which options have been selected. If the driver uses memory at an improper IRQL, improperly calls or releases spin locks and memory allocations, improperly switches stacks, or frees memory pool without first removing timers, Driver Verifier will detect this behavior. When the driver is unloaded, Driver Verifier will check to see that it has properly released its resources.

- **Special Memory Pool**

When this option is active, Driver Verifier allocates most of the driver's memory requests from a special pool. This special pool is monitored for memory overruns, memory underruns, and memory that is accessed after it is freed.

- **Forcing IRQL Checking**

When this option is active, Driver Verifier places extreme memory pressure on the driver by invalidating pageable code. If the driver attempts to access paged memory at the wrong IRQL or while holding a spin lock, Driver Verifier detects this behavior.

- **Low Resources Simulation**

When this option is active, Driver Verifier randomly fails pool allocation requests and other resource requests. By injecting these allocation faults into the system, Driver Verifier tests the driver's ability to cope with a low-resource situation.

- **Memory Pool Tracking**

When this option is active, Driver Verifier checks to see if the driver has freed all its memory allocations when it is unloaded. This reveals memory leaks.

- **I/O Verification**

When this option is active, Driver Verifier allocates the driver's IRPs from a special pool, and monitors the driver's I/O handling. This detects illegal or inconsistent use of I/O routines.

- **DMA Verification**

(Windows XP and later) When this option is active, Driver Verifier monitors the driver's use of DMA routines. This detects improper use of DMA buffers, adapters, and map registers.

- **Deadlock Detection**

(Windows XP and later) When this option is active, Driver Verifier monitors the driver's use of spin locks, mutexes, and fast mutexes. This detects if the driver's code has the potential for causing a deadlock at some point.

- **SCSI Verification**

(Windows XP and later) When this option is active, Driver Verifier monitors a SCSI miniport driver for improper use of exported SCSI port routines, excessive delays, and improper handling of SCSI requests.

- **Disk Integrity Verification**

(Windows .NET Server and later) When this option is active, Driver Verifier monitors hard disk access, and detects whether the disk is preserving its data correctly.

Driver Verifier can verify any number of drivers. However, the Special Memory Pool and I/O Verification options will be more effective when used on one driver at a time.

Automatic Checks

Driver Verifier performs the following actions whenever it is verifying one or more drivers. These actions are not affected by enabling or disabling any of Driver Verifier's options.

Monitoring IRQL and Memory Routines

Driver Verifier monitors the selected driver for the following forbidden actions:

- Raising IRQL by calling **KeLowerIrql**
- Lowering IRQL by calling **KeRaiseIrql**
- Requesting a size zero memory allocation
- Allocating or freeing paged pool at an IRQL above APC_LEVEL
- Allocating or freeing nonpaged pool at an IRQL above DISPATCH_LEVEL
- Trying to free an address that was not returned from a previous allocation
- Trying to free an address that was already freed
- Acquiring or releasing a fast mutex at an IRQL above APC_LEVEL
- Acquiring or releasing a spin lock at an IRQL other than DISPATCH_LEVEL
- Double-releasing a spin lock.
- Specifying an illegal or random (uninitialized) parameter to any one of several APIs.
- Marking an allocation request MUST_SUCCEED. No such requests are ever permissible.

If Driver Verifier is not active, these violations might not cause an immediate system crash in all cases. Driver Verifier monitors the driver's behavior and issues bug check 0xC4 if any of these violations occur. See Bug Check 0xC4 (DRIVER_VERIFIER_DETECTED_VIOLATION) in the Debugging Tools for Windows documentation for a list of the bug check parameters.

Monitoring Stack Switching

Driver Verifier monitors stack usage by the driver being verified. If the driver switches its stack, and the new stack is neither a thread stack nor a DPC stack, then a bug check is issued. (This will be bug check 0xC4 with the first parameter equal to 0x90.) The stack displayed by the **KB** debugger command will usually reveal the driver that performed this operation.

Checking Freed Pool for Timers

Driver Verifier examines all memory pool freed by the driver being verified. If any timers remain in this pool, bug check 0xC7 is issued. (Forgotten timers can eventually lead to system crashes that are notoriously difficult to account for.)

Checking on Driver Unload

After a driver that is being verified unloads, Driver Verifier performs several checks to make sure that the driver has cleaned up.

In particular, Driver Verifier looks for:

- Undeleted timers
- Pending deferred procedure calls (DPCs)
- Undeleted lookaside lists
- Undeleted worker threads
- Undeleted queues
- Other similar resources

Problems such as these can potentially cause system bug checks to be issued a while after the driver unloads, and the cause of these bug checks can be hard to determine. When Driver Verifier is active, such violations will result in bug check 0xC7 being issued immediately after the driver is unloaded. See Bug Check 0xC7 (TIMER_OR_DPC_INVALID) in the Debugging Tools for Windows documentation for a list of the bug check parameters.

Driver Verifier Options

- **Special Memory Pool**
 - Used to detect memory overwrites and underwrites
- **Memory Pool Tracking**
 - Verifies the driver freed all allocated memory on exit
- **Forcing IRQL Checking**
 - Catches the driver when it touches paged code at a raised IRQL or while holding a spin lock
- **Low Resources Simulation**
 - Randomly fails allocation calls for device drivers being verified

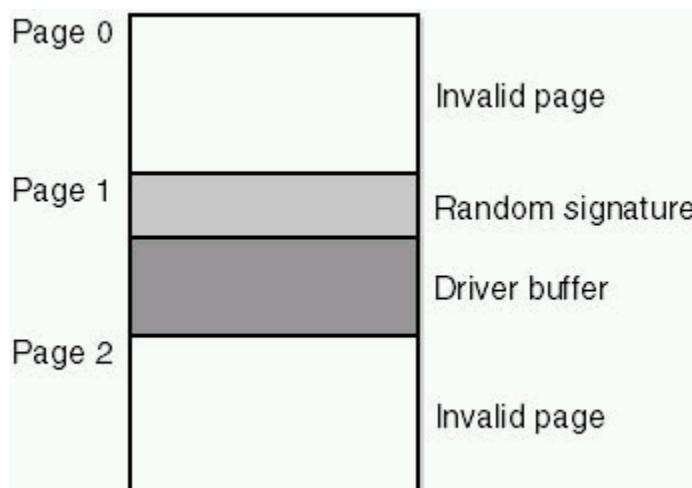
Special Memory Pool

Memory corruption is a common driver problem. Driver errors can result in crashes long after the errors are made. The most common of these errors is accessing memory that has already been freed, and allocating n bytes and then accessing $n+1$ bytes.

To detect memory corruption, Driver Verifier can allocate driver memory from a special pool and monitor that pool for incorrect access.

Two alignments of the special pool are available. The **Verify End** alignment is better at detecting access overruns, and the **Verify Start** alignment is better at detecting access underruns. (Note that the vast majority of memory corruptions are due to overruns, not underruns.)

When Special Memory Pool is active and **Verify End** has been selected, each memory allocation requested by the driver is placed on a separate page. The highest possible address that allows the allocation to fit on the page is returned, so that the memory is aligned with the end of the page. The previous portion of the page is written with special patterns. The previous page and the next page are marked inaccessible. This is illustrated in the following figure:



If the driver attempts to access memory after the end of the allocation, Driver Verifier will detect this immediately, and will issue bug check 0xCD. If the driver writes in the memory prior to the beginning of the buffer, this will (presumably) alter the patterns. When the buffer is freed, Driver Verifier will detect the alteration and issue bug check 0xC1.

If the driver reads or writes to the buffer after freeing it, Driver Verifier will issue bug check 0xCC.

When **Verify Start** is selected, the memory buffer is aligned with the beginning of the page. With this setting, underruns cause an immediate bug check and overruns cause a bug check when the memory is freed. This option is otherwise identical to the **Verify End** option.

Verify End is the default alignment, as overrun errors are much more common in drivers than underrun errors.

An individual memory allocation can override these settings and choose its alignment by calling **ExAllocatePoolWithTagPriority** with the *Priority* parameter set to `XxxSpecialPoolOverrun` or `XxxSpecialPoolUnderrun`. (This routine cannot activate or deactivate the special pool, or request the special pool for a memory allocation, which would otherwise be allocated from normal pool. Only the alignment can be controlled from this routine.)

Special Pool by Pool Tag or Allocation Size

There are two other ways to use the special pool: it can be used for all allocations that are marked with a specific pool tag, or for all allocations whose size is within a specific size range. These uses of special pool are controlled through the Global Flags utility, not Driver Verifier.

It is permissible to request special pools though Driver Verifier and the Global Flags utility at the same time. In this case, Windows will attempt to use the special pool for all allocations indicated by Driver Verifier *and* all allocations indicated by the Global Flags utility.

Special Pool Efficiency

Each allocation from the special pool uses one page of nonpageable memory and two pages of virtual address space. If the pool is exhausted, memory is allocated in the standard way until the special pool becomes available again. Thus, it is not recommended that multiple drivers be verified at the same time if Special Memory Pool is in effect.

A single driver that makes a large number of small memory requests can also deplete this pool. If this is occurring, it may be preferable to assign pool tags to the driver's memory allocations and dedicate the special pool to one pool tag at a time.

The size of the special pool increases with the amount of physical memory on the system; ideally this should be at least 1 Gigabyte (GB). On x86 machines, booting without the **/3GB** switch is also preferred, as virtual (in addition to physical) space is consumed. Increasing the pagefile minimum/maximum quantities (by a factor of two or three) is also a good idea.

To be sure that all of a driver's allocations are being tested, stressing the driver over long periods of time is recommended.

Monitoring the Special Pool

Statistics relating to pool allocations can be monitored. These can be displayed by the Driver Verifier Manager graphical interface, the *Verifier.exe* command line, or in a log file.

If the *Pool Allocations Succeeded in Special Pool* counter is equal to the *Pool Allocations Succeeded* counter, then the special pool has been sufficient to cover all memory allocations. If the former counter is lower than the latter, then the special pool has been exhausted at least once.

These counters do not track allocations whose size is one page or larger, since the special pool is not applicable to them.

If Special Memory Pool is enabled, but less than 95% of all pool allocations have been assigned from the special pool, a warning will appear in the Driver Verifier Manager graphical interface. In Windows 2000, this warning will appear on the **Driver Status** screen. In Windows XP and later, this warning will appear on the **Global Counters** screen. If this occurs, you should verify a shorter list of drivers, verify individual pools by pool tag, or add more physical memory to your system.

The kernel debugger extension **!Verifier** can also be used to monitor special pool use. It presents similar information to that of Driver Verifier Manager. For details, see the Debugging Tools for Windows documentation.

Forcing IRQL Checking

Although kernel-mode drivers are forbidden to access pageable memory at a high IRQL or while holding a spin lock, such an action might not be noticed if the page has not actually been trimmed.

When Forcing IRQL Checking is enabled, Driver Verifier will provide extreme memory pressure on the selected driver. Whenever a driver being verified requests a spin lock, calls **KeSynchronizeExecution**, or raises the IRQL to DISPATCH_LEVEL or higher, all of the driver's pageable code and data (as well as system pageable pool, code, and data) are marked as trimmed. If the driver attempts to access any of this memory, Driver Verifier issues a bug check.

This memory pressure will not directly affect drivers that are not selected for verification, since other drivers' IRQL raises will not cause this action. However, when a driver that is being verified raises the IRQL, Driver Verifier trims pages that can be used by drivers that are *not* being verified. Thus errors by drivers that are not being verified might occasionally be caught when this option is active.

Monitoring IRQL Raises and Spin Locks

The number of IRQL raises, spin locks, and calls to **KeSynchronizeExecution** made by drivers being verified can be monitored. The number of pageable memory trims forced by Driver Verifier can also be monitored. These statistics can be displayed by the Driver Verifier Manager graphical interface, the *Verifier.exe* command line, or in a log file.

The kernel debugger extension **!Verifier** can also be used to monitor these statistics. It presents similar information to that of Driver Verifier Manager. In Windows XP and later, the **!Verifier 0x8** extension will display a log of recent IRQL changes made by drivers being verified. For details, see the Debugging Tools for Windows documentation.

Low Resources Simulation

When Low Resources Simulation is enabled, Driver Verifier will cause a random selection of the driver's memory allocations to fail. This tests the driver's ability to react properly to low-memory and other low-resource conditions.

To accurately simulate a low-memory condition, these allocation faults are not injected until seven minutes after system startup. Therefore, any driver errors that are exposed by this action should be treated as legitimate run-time problems, not as unrealistic scenarios.

Driver Verifier can verify selected drivers or all drivers at the same time.

Monitoring the Low Resources Simulation

The number of times Driver Verifier deliberately fails resource allocations can be monitored. This statistic can be displayed by the Driver Verifier Manager graphical interface, the *Verifier.exe* command line, or in a log file.

The kernel debugger extension **!Verifier** can also be used to monitor this statistic. It presents similar information to that of Driver Verifier Manager. In Windows XP and

later, the **!Verifier 0x4** extension will display a log of faults injected by Driver Verifier. For details, see the Debugging Tools for Windows documentation.

Memory Pool Tracking

Memory Pool Tracking monitors the memory allocations made by the driver. At the time that the driver is unloaded, Driver Verifier ensures that all allocations made by the driver have been freed.

Unfreed memory allocations (also called *memory leaks*) are a common cause of lowered operating system performance. These can fragment the system pools and eventually cause system crashes.

When this option is active, Driver Verifier will issue bug check 0xC4 (with Parameter 1 equal to 0x60) if a driver unloads without freeing all its allocations.

If Driver Verifier issues this bug check with Parameter 1 equal to 0x51, 0x52, 0x53, 0x54, or 0x59, the driver has written to memory outside of its allocations. In this case, you should enable Special Memory Pool to locate the source of the error.

See Bug Check 0xC4 (DRIVER_VERIFIER_DETECTED_VIOLATION) in the Debugging Tools for Windows documentation for a list of the bug check parameters.

Monitoring Memory Pool Tracking

Memory pool allocation statistics can be monitored separately for each driver being verified. These statistics can be displayed by the Driver Verifier Manager graphical interface, the *Verifier.exe* command line, or in a log file.

The kernel debugger extension **!Verifier 0x3** can be used to locate outstanding memory allocations after the driver is unloaded, or to track the current allocations while the driver is running. This extension also shows the pool tag, the size of the pool, and the address of the allocator for each allocation. For details, see the Debugging Tools for Windows documentation.

Driver Verifier Options – Continued

- **I/O Verification**
 - Uses special IRP pool and monitors the driver's I/O use
- **Deadlock Detection**
 - (*New for Windows XP*) Catches the driver when it touches paged code at a raised IRQL or while holding a spin lock
- **DMA Verification**
 - (*New for Windows XP*) Monitors use of Direct Memory Access
- **SCSI Verification**
 - (*New for Windows XP*) Monitors SCSI miniport activity
- **Disk Integrity Verification**
 - (*New for Windows .NET Server*) Verifies that disk data doesn't change between disk access operations

I/O Verification

Driver Verifier has three levels of I/O Verification:

- *Level 1 I/O Verification* is always active whenever I/O Verification is selected.
- *Level 2 I/O Verification* is always active whenever I/O Verification is selected in Windows XP and later. In Windows 2000, I/O Verification can be configured to include both levels, or just the Level 1 tests.
- *Enhanced I/O Verification* can be activated separately from the basic two I/O Verification levels in Windows XP and later. It is not available in Windows 2000.

The effects of each level will be described separately.

Level 1 I/O Verification

When Level 1 I/O Verification is enabled, all IRPs obtained through **IoAllocateIrp** are allocated from a special pool and their use is tracked.

Additionally, Driver Verifier checks for invalid I/O calls, including:

- Attempts to free an IRP whose type is not IO_TYPE_IRP
- Passes of invalid device objects to **IoCallDriver**
- Passes of an IRP to **IoCompleteRequest** that contains invalid status or that still has a cancel routine set
- Changes to the IRQL across a call to the driver dispatch routine
- Attempts to free an IRP that remains associated with a thread
- Passes of a device object to **IoInitializeTimer** that already contains an initialized timer
- Passes of an invalid buffer to **IoBuildAsynchronousFsdRequest** or **IoBuildDeviceIoControlRequest**
- Passes of an I/O status block to an IRP, when this I/O status block is allocated on a stack that has unwound too far
- Passes of an event object to an IRP, when this event object is allocated on a stack that has unwound too far

Because the special IRP pool is of limited size, I/O Verification is most effective when it is only used on one driver at a time.

I/O Verification Level 1 failures cause bug check 0xC9 to be issued. The first parameter of this bug check indicates what violation has occurred. See Bug Check 0xC9 (DRIVER_VERIFIER_IOMANAGER_VIOLATION) in the Debugging Tools for Windows documentation for a full parameter listing.

Level 2 I/O Verification

I/O Verification Level 2 errors are displayed in different ways: on the blue screen, in a crash dump file, and in a kernel debugger.

On the blue screen, these errors are noted by the message **IO SYSTEM VERIFICATION ERROR** and the string **WDM DRIVER ERROR XXX**, where *XXX* is an I/O error code.

In a crash dump file, these errors are noted by the message **BugCheck 0xC9 (DRIVER_VERIFIER_IOMANAGER_VIOLATION)**, along with the I/O error code. In this case, the I/O error code appears as the first parameter of the bug check 0xC9.

In a kernel debugger (KD or WinDbg), these errors are noted by the message **WDM DRIVER ERROR** and a descriptive text string. When the kernel debugger is active, it is possible to ignore the Level 2 errors and resume system operation. (This is not possible with any other bug checks.)

The blue screen, the crash dump file, and the kernel debugger each display additional information as well. For a full description of all I/O Verification Level 2 error messages, see Bug Check 0xC9 in the Debugging Tools for Windows documentation.

Enhanced I/O Verification

This option is only available in Windows XP and later.

The Enhanced I/O Verification option includes some new parameter verifications, as well as some new stresses and tests.

New parameter verifications include:

- Monitoring all IRPs for correct use of **IoMarkPending**. (Drivers should return STATUS_PENDING if and only if they have called **IoMarkPending**.)
- Monitoring the use of **IoDeleteDevice**, to catch double-deletions and inappropriate detaching and deleting of device objects.
- Unwinding all **IoSkipCurrentIrpStackLocation** calls.

New stresses and tests include:

- Scrambling the order of enumerated devices, to ensure that Plug and Play (PnP) drivers don't make assumptions about the device start order.
- Adjusting the status of PnP and Power IRPs when they complete, to catch drivers that return an incorrect status from their dispatch routines.
- Sending fake Power IRPs to test for driver code path bugs.
- Sending fake WMI IRPs to test for driver code path bugs.
- Inserting a fake filter into every WDM stack.

Driver errors caught by Enhanced I/O Verification are displayed in the same manner as those caught by Level 2 I/O Verification.

Deadlock Detection

Deadlock Detection monitors the driver's use of resources which need to be locked — spin locks, mutexes, and fast mutexes. This Driver Verifier option will detect code logic that has the potential to cause a deadlock at some future point.

The Deadlock Detection option of Driver Verifier, along with the **!deadlock** kernel debugger extension, is an effective tool for making sure your code avoids poor use of these resources.

This Driver Verifier option is only available in Windows XP and later.

Causes of Deadlocks

A *deadlock* is caused when two or more threads come into conflict over some resource, in such a way that no execution is possible.

The most common form of deadlock occurs when two or more threads wait for a resource that is owned by the other thread. This is illustrated as follows:

Thread 1	Thread 2
Takes Lock A	Takes Lock B
Requests Lock B	Requests Lock A

If both sequences happen at the same time, Thread 1 will never get Lock B because it is owned by Thread 2, and Thread 2 will never get Lock A because it is owned by Thread 1. At best this causes the threads involved to halt, and at worst causes the system to stop responding.

Deadlocks are not limited to two threads and two resources. Three-way deadlocks between three threads and three locks are common — and even five-part or six-part deadlocks occur occasionally. These deadlocks require a certain degree of "bad luck" since they rely on a number of things happening simultaneously. However, the farther apart the lock acquisitions are, the more likely these become.

Single-thread deadlocks can occur when a thread attempts to take a lock that it already owns.

The common denominator among all deadlocks is that *lock hierarchy is not respected*. Whenever it is necessary to have more than one lock acquired at a time, each lock should have a clear precedence. If A is taken before B at one point and B before C at another, the hierarchy is A-B-C. This means that A must never be acquired after B or C, and B must not be acquired after A.

Lock hierarchy should be followed *even when there is no possibility of a deadlock*, since in the process of maintaining the code it will be easy for a deadlock to be accidentally introduced.

Resources That Can Cause Deadlocks

The most unambiguous deadlocks are the result of *owned* resources. These include spin locks, mutexes, fast mutexes, and ERESOURCES.

Resources that are signaled rather than acquired (such as events and LPC ports) tend to cause much more ambiguous deadlocks. It is of course possible, and all too common, for code to misuse these resources in such a way that two threads will end up waiting on each other indefinitely. However, since these resources are not actually owned by any one thread, it is not possible to identify the delinquent thread with any degree of certainty.

The Deadlock Detection option of Driver Verifier looks for potential deadlocks involving spin locks, mutexes, and fast mutexes. It does not monitor the use of ERESOURCES, nor does it monitor the use of non-owned resources.

Effects of Deadlock Detection

Driver Verifier's Deadlock Detection routines find lock hierarchy violations that are not necessarily simultaneous. Most of the time, these violations identify code paths that will deadlock when given the chance.

To find potential deadlocks, Driver Verifier builds a graph of resource acquisition order and checks for loops. If you were to create a node for each resource, and draw an arrow any time one lock is acquired before another, then path loops would represent lock hierarchy violations.

Driver Verifier will issue a bug check when one of these violations is discovered. This will happen *before* any actual deadlocks occur.

Note: Even if the conflicting code paths can never happen simultaneously, they should still be rewritten if they involve lock hierarchy violations. Such code is a "deadlock waiting to happen" that could cause real deadlocks if the code is rewritten slightly.

When Deadlock Detection finds a violation, it will issue bug check 0xC4. The first parameter of this bug check will indicate the exact violation. Possible violations include:

- Two or more threads involved in a lock hierarchy violation
- A resource that is released out of sequence
- A thread that tries to acquire the same resource twice (a self-deadlock)
- A resource that is released without having been acquired first
- A resource that is released by a different thread than the one that acquired it
- A resource that is initialized more than once, or not initialized at all
- A thread that is deleted while still owning resources

See Bug Check 0xC4 (DRIVER_VERIFIER_DETECTED_VIOLATION) in the Debugging Tools for Windows documentation for a list of the bug check parameters.

Monitoring Deadlock Detection

Once Deadlock Detection finds a violation, the **!deadlock** kernel debugger extension can be used to investigate exactly what has occurred. It can display the lock hierarchy topology as well as the call stacks for each thread at the time the locks were originally acquired.

For best results, the driver in question should be running on a checked build of Windows, since that allows the kernel to obtain more complete run-time stack traces.

For information about the **!deadlock** extension, see the Debugging Tools for Windows documentation.

DMA Verification

DMA Verification monitors the use of Direct Memory Access (DMA). Since the DMA routines have changed as Windows has developed, many drivers make incorrect use of DMA calls. Moreover, some driver writers attempt to bypass the HAL DMA subsystem altogether. This practice can introduce insidious bugs into the driver.

The DMA Verification option of Driver Verifier attempts to catch common DMA errors. Along with the **!dma** kernel debugger extension, it can be used to verify that a driver is using DMA in a proper manner.

This Driver Verifier option is also called **HAL Verification**. Some error messages produced by Driver Verifier may use this term.

This Driver Verifier option is only available in Windows XP and later.

Different Types of DMA

DMA is a mechanism through which a hardware device can transfer data to or from memory without using the processor. The processor is required to set up the transfer, and the device will signal the processor when it has completed the transfer. The advantage of this system is that the processor can perform other tasks while the DMA transfer is being performed.

There are several types of DMA used in Windows 2000 and later:

Common-buffer DMA

Common-buffer DMA is performed when the system can allocate a single buffer that is accessible by both the hardware and the software. The driver is responsible for synchronizing accesses to the buffer. The memory is not cached, making this synchronization easier for the driver. After setting up a common buffer, both the

driver and the hardware can write directly to the addresses in the buffer without any intervention from the HAL.

Packet DMA

Packet DMA is performed when there is a single existing buffer that must be mapped for use by the hardware. An example of using packet DMA is the transfer of a file from memory to a disk. Using common-buffer DMA in this situation would be wasteful, because the file would have to be transferred to the common buffer before the hardware could transfer it to the disk. Instead, the HAL is consulted; it gives the driver the information it needs to help the hardware find the actual buffer in memory. This operation is complicated by the need for the routines involved to work across different architectures.

Scatter-Gather DMA

Scatter-gather DMA is a shortcut method that sets up several packet DMA transfers at once. If you are transferring a packet over the network, for example, each part of the network stack adds its own header (TCP, IP, Ethernet, and so forth). These headers are all allocated from different places in memory. In this case, the scatter-gather DMA saves time by issuing a batch request to the HAL to map each header plus the data segment for access by the hardware. Instead of having to call the packet DMA routines on each part of the packet, this method calls each routine once, and lets the HAL be responsible for mapping each one individually.

System DMA

System DMA is performed by programming the system DMA controller on the motherboard to do the transfer directly. Only ISA cards can use system DMA.

Note: *Scatter-gather capability* does not mean that the device can use the scatter-gather routines. Scatter-gather capability refers to a flag in the device description that indicates that the device is able to read or write from any area in memory, instead of just a certain range.

Effects of DMA Verification

When DMA Verification is active, the following common misuses of the DMA routines are detected:

- Overrunning the DMA memory buffer (these overruns can be made by the hardware or the driver)
- Attempting to use an adapter that was already freed and no longer exists
- Double-freeing a common buffer, adapter channel, map register, or scatter-gather list
- Leaking memory by not freeing common buffers, adapter channels, map registers, scatter-gather lists, or adapters
- Having more than one adapter channel present for an adapter at one time
- Allocating too many map registers
- Not flushing adapter buffers
- Double-mapping of map registers
- Performing DMA on a pageable buffer (all buffers should be locked before DMA transfer begins)
- Performing DMA on an MDL with mangled flags
- Calling DMA routines at an improper IRQL

Driver Verifier monitors the driver's behavior and issues bug check 0xE6 if any of these violations occur. See Bug Check 0xE6 (DRIVER_VERIFIER_DMA_VIOLATION) in the Debugging Tools for Windows documentation for a list of the bug check parameters.

When is DMA Verification Useful?

All drivers that use DMA directly (by calling the HAL DMA routines) should be tested with DMA Verification.

In addition, miniport drivers should also be tested, since they often use DMA indirectly (by calling port drivers that use DMA).

DMA Verification can also be an effective way to detect memory corruption, since it can spot when either a driver or a hardware device overruns a DMA buffer.

Monitoring DMA Verification

The kernel debugger extension **!dma** can be used to display a wealth of DMA information. It can display various details about the behavior of each DMA adapter. For information, see the Debugging Tools for Windows documentation.

SCSI Verification

The SCSI Verification feature of Driver Verifier monitors the interaction between a SCSI miniport driver and the port driver. If the miniport driver misuses a routine, responds incorrectly to a request from the port driver, or takes an excessive amount of time to respond to a request, a bug check is issued.

This Driver Verifier option is only available in Windows XP and later.

Violations Detected by SCSI Verification

The SCSI Verification option can detect several misuses of SCSI routines. It is also possible to individually disable certain of these checks.

When a SCSI miniport driver commits one of the following violations, Driver Verifier will issue bug check 0xF1.

- The miniport passes a bad argument to **ScsiPortInitialize**.
- The miniport calls **ScsiPortStallExecution** and specifies a delay longer than 0.1 second, stalling the processor for an excessive length of time.
- The port driver calls a miniport routine, and the miniport takes longer than 0.5 second to execute it. (The **FindAdapter** routine is exempt, and the **HwInitialize** routine is allowed 5 seconds.)
- The miniport completes a request more than once.
- The miniport completes a routine with an invalid SRB status.
- The miniport calls **ScsiPortNotification** to ask for **NextLuRequest**, but an untagged request is still active.
- The miniport passes an invalid virtual address to **ScsiPortGetPhysicalAddress**. (This usually means the address supplied doesn't map to the common buffer area.)
- The bus reset hold period ends, but the miniport still has outstanding requests.

See Bug Check 0xF1 (SCSI_VERIFIER_DETECTED_VIOLATION)) in the Debugging Tools for Windows documentation for a complete list of the bug check parameters.

In addition to these violations, SCSI Verification also monitors the miniport driver's memory access for improper use. Two common memory violations made by miniport drivers are accessing an SRB extension after a request completes, and accessing an SRB's **DataBuffer** when the miniport has not specified **MapBuffers**.

Memory violations of this sort will usually result in Bug Check 0xD1 (DRIVER_IRQL_NOT_LESS_OR_EQUAL) being issued.

Activating This Option

The SCSI Verification option is enabled in a different manner from other Driver Verifier options.

1. Using the Driver Verifier Manager graphical interface or the verifier.exe command line, request verification of the desired miniport driver. SCSI Verification will not be available as an option, but you must select at least one other Driver Verifier option.

2. Open the registry using *regedit.exe*. In the *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\ScsiPort* key, add a subkey named **Verifier**. Within that key, add a REG_DWORD entry named **VerifyLevel**. The value assigned to this entry will determine which SCSI Verification tests will be active. The value 0x1 will give maximum verification.
3. Reboot the computer.

If the **VerifyLevel** value does not exist, or is equal to 0xFFFFFFFF, SCSI Verification will be disabled.

The individual bits in the **VerifyLevel** value can be used to control exactly which tests will be performed. Bit zero (0x1) enables certain tests; bits 28, 29, 30, and 31 *disable* certain tests. Therefore, maximum verification can be obtained by using the value 0x00000001.

The effects of each bit are as follows:

Bit	Value	Effect
0	0x1	Driver Verifier will monitor the miniport driver's memory access and check for improper use of memory buffers.
28	0x10000000	Driver Verifier will not issue a bug check when the HwAdapterControl routine takes more than 0.5 second to complete.
29	0x20000000	Driver Verifier will not issue a bug check when a reset hold period ends and there are still outstanding requests on a logical unit.
30	0x40000000	Driver Verifier will not issue a bug check when the miniport calls ScsiPortNotification with NextLuRequest while an untagged request is still active.
31	0x80000000	Driver Verifier will not issue a bug check when the HwInitialize routine takes more than 5 seconds to complete.

In most cases, the recommended setting is 0xD0000001. This enables all **SCSI Verifier** tests except for the time limit on **HwAdapterControl**, the time limit on **HwInitialize**, and the ban on multiple requests to a logical unit. These three tests are often too stringent.

If a kernel debugger is attached, it is possible to change the SCSI Verification level *after* the boot cycle. To do this, use the debugger command:

```
kd> ed scsiprt!SpVrfyLevel Level
```

This command allows you to set a new value for *Level*. Using this method, you can change the high bits (0x10000000 through 0x80000000) at any time. However, if you wish to change the low bit (0x1), you must do so during the boot process (at the kernel debugger's initial breakpoint).

Similarly, if you want to completely deactivate SCSI Verification, you need to set *Level* to 0xFFFFFFFF at the initial breakpoint.

Note: The value 0xF0000000 will disable all tests, but the SCSI Verification modules will still be loaded. Use this value if you wish to disable verification but intend to enable the high-bit tests at a later time. On the other hand, the value 0xFFFFFFFF prevents the modules from being loaded entirely; if this value is used during boot it will not be possible to enable SCSI Verification without rebooting.

Disk Integrity Verification

The Disk Integrity Verification feature of Driver Verifier monitors all hard disk access to determine whether the disk accurately stores information. If the data on the disk appears to change, a bug check is issued.

This Driver Verifier option is only available in Windows .NET Server and later.

How Disk Integrity Verification Works

When you activate Disk Integrity Verification, you can choose to verify any or all of the physical disks attached to your computer.

As soon as Windows and its drivers are loaded, Driver Verifier begins to monitor all read and write actions made to these drives. Driver Verifier calculates a CRC (cyclic redundancy check) checksum value for each sector that is accessed and saves this value. The next time this sector is accessed, Driver Verifier recalculates this checksum and compares it to the previous value.

If the checksum value changes, this indicates a disk integrity problem — either the read operation is returning faulty information, or the disk medium has altered its contents since the last access was made. When this happens, Driver Verifier issues bug check 0xC4 with Parameter 1 equal to 0xA0. The other parameters identify the IRP making the request, the device object of the lower device, and the sector in which the error occurred. For details, see Bug Check 0xC4 (DRIVER_VERIFIER_DETECTED_VIOLATION) in the Debugging Tools for Windows documentation.

Performance Issues

When Disk Integrity Verification is active, performance is adversely affected whenever the hard disk is accessed. If your computer is low on RAM, this performance decrease is even more significant. You should use Disk Integrity Verification whenever you are investigating disk problems, but you should not gratuitously activate it when you are running Driver Verifier to test drivers.

Module 10 Labs



Module 11: User-Mode Debugging

Module 11 Overview

- Configuring for User Mode Debugging
- Starting the Debugger
- Using the Debugger Command Window
- Choosing Processes and Threads
- Basic Debugger Operations
- User Mode Dump Files
- Debugging Memory Leaks
- Debugging User-Mode System Processes

Configuring and Starting the Debugger

Configuring Software for User-Mode Debugging

• Basic User-Mode Configuration

- Install the Debugger
- Install the Symbol files (or plan to use symbol server)
- Configure Environment Variables

Before you can begin user-mode debugging, you must install the necessary symbol files and set certain environment variables.

Installing the Symbol Files

You must install the symbol files for the user-mode process that is being debugged. If this is an application you have written, it should be built with full symbol files. If it is a commercial application, the symbol files may be on the product disk; if not, contact the manufacturer.

You should also install the symbol files for the version of Windows® on which the user-mode process is running.

Configuring Environment Variables

The debugger uses a variety of environment variables to indicate a number of important settings. For a complete list of environment variables used in user-mode debugging, refer to the debugger documentation.

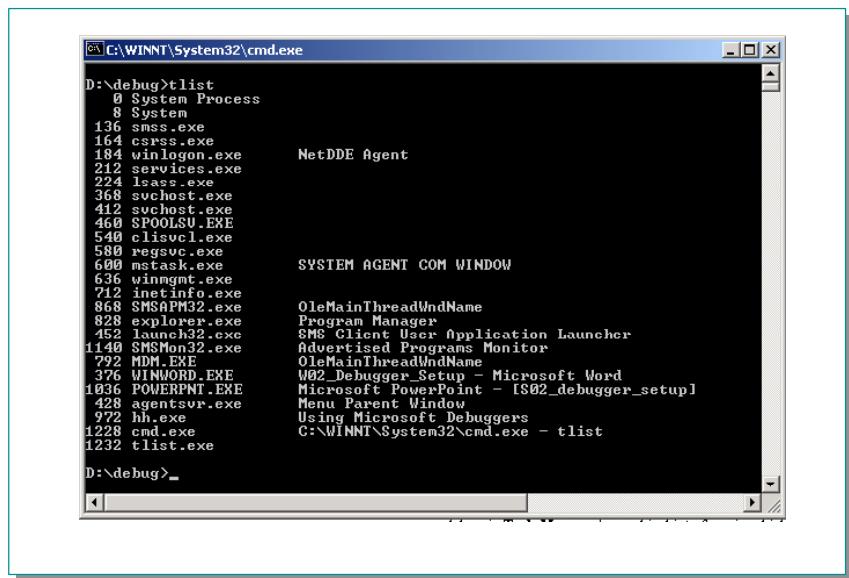
Starting the Debugger

- Attaching to a Running Process (User Mode)
- Spawning a New Process (User Mode)

When you start WinDbg, there are several ways you can select your debugging target. You can attach to a process that is already running, or you can spawn a new instance of a program and use it as your target application.

There are two other possible ways to begin a debugging session. You can join a remote debugging session by connecting to a Debugging Server. Or the debugger can also be automatically activated when a user-mode process fails by enabling “postmortem debugging” (also known as “just-in-time debugging,” or “JIT debugging.”)

Using TList to get a Process ID (PID)



Finding the Process ID

Each process running in Microsoft® Windows® is assigned a unique decimal number called the *process ID*, or *PID*.

If you wish to attach to a process that is already running, you need to determine its process ID.

There are several ways to determine the PID for a given application: using the Task Manager, using the **tasklist** command, using the TList utility, or using the debugger.

Task Manager

The *Task Manager* may be activated in a number of ways, but the simplest is to press CTRL+ALT+DELETE and then click on **Task Manager**.

If you select the **Processes** tab, each process and its PID will be listed, along with other useful information.

Some kernel errors may cause delays in **Task Manager**'s graphical interface.

The Tasklist Command

In Windows XP and Windows .NET Server 2003, you can use the **tasklist** command from a Command Prompt window. This displays all processes, their PIDs, and a variety of other details.

TList

TList (*Task List Viewer, tlist.exe*) is a command-line utility that displays a list of tasks, or user-mode processes, currently running on the local computer. TList is included in the Debugging Tools for Windows package.

When you run TList from the command prompt, it will display a list of all the user-mode processes in memory with a unique process identification (PID) number. For each process, it shows the PID, process name, and, if the process has a window, the title of that window.

The .tlist Debugger Command

If there is already a user-mode debugger running on the system in question, the **.tlist** command will display a list of all PIDs on that system.

Attaching to a Running Process

- **NTSD Command Line**
 - `ntsd -p pid`
- **CDB Command Line**
 - `cdb -p pid`
- **WinDbg Command Line**
 - `windbg -p pid`
- **WinDbg Menu**
 - **File | Attach to Process | Select Process**

If a user-mode application is already running, the debugger can attach to that process and then debug it. The application is specified by a *process ID* (PID).

WinDbg Command Line

To attach to a running process from the WinDbg command line, specify the **-p** option and the process ID:

```
windbg -p pid
```

WinDbg Menu

When WinDbg is in dormant mode, you can attach to a running process by selecting the **File | Attach to a Process** menu command or by pressing the F6 shortcut key.

When the **Attach to Process** dialog box appears, enter the process ID in the text box and press **OK**.

Spawning a New Process

- **NTSD Command Line**
 - `ntsd [-o] ProgramName [Arguments]`
- **CDB Command Line**
 - `cdb [-o] ProgramName [Arguments]`
- **WinDbg Command Line**
 - `windbg [-o] ProgramName [Arguments]`
- **WinDbg Menu**
 - File | Open Executable

The debugger can start a user-mode program and then debug it. The application is specified by name, just as it would be if launched from a Microsoft® Windows® NT Command Prompt.

The debugger can also automatically attach to *child processes* (additional processes launched by the original target process).

WinDbg Command Line

To spawn a user-mode program from the WinDbg command line, place the program's own command line at the end of the WinDbg command line:

`windbg [-o] ProgramName [Arguments]`

The `-o` option will cause the debugger to attach to child processes.

WinDbg Menu

When WinDbg is in dormant mode, you can spawn a new process by selecting the **File | Open Executable** menu command or by pressing the CTRL+E shortcut key.

When the **Open Executable** dialog box appears, enter the full path and name of the executable in the **File name** text box, or use the dialog box to select the proper path and file name. (You need to specify the exact path to this program. Whereas the Windows® **Start | Run** dialog box and the Command Prompt window search the command path for an executable name, the **Open Executable** dialog box does not.)

If you wish to use any command-line arguments with this application, type them in the **Arguments** text box.

If you wish WinDbg to attach to child processes, select the **Debug child processes also** check box.

When the proper selections have been made, press **Open**.

Basic Debugger Operations

The Debugger Command Window

- **Using Debugger Commands**
 - **The Debugger Command Window Prompt**
 - 2:005> (current process and thread IDs)

The Debugger Command Window Prompt

When performing user-mode debugging, the prompt looks like this:

2:005>

In this example, "2" represents the current process ID and "005" represents the current thread ID.

If no prompt appears, or if the entry pane in WinDbg's Debugger Command window is grayed, no command will be accepted. However, menu commands and shortcut keys can still be used.

Editing, Repeating, and Canceling Commands

You can use standard editing keys when you enter a command. Use the UP and DOWN ARROW keys to retrieve previous commands. Edit the current command line with the BACKSPACE, DEL, INS, and LEFT and RIGHT ARROW keys. Press the ESC key to clear the current line.

Pressing the ENTER key by itself will repeat the previous command. (This behavior can be enabled or disabled.)

If the last command issued is presenting a lengthy display and you wish to cut it off, use **Debug | Break** or press the CTRL+BREAK shortcut key.

Clearing Command Output

You can use the **.cls (Clear Screen)** command to clear all the text from the Debugger Command window, clearing the command history entirely.

This is equivalent to the **Edit | Clear Command Output** menu command.

Choosing Processes and Threads

- **Display Process and Thread Information**
 - **| (Process Status)**
 - **~ (Thread Status)**
- **Change the Current Process or Thread**
 - **|S (Set Current Process)**
 - **~S (Set Current Thread)**
- **Freeze and Unfreeze Individual Threads**
 - **~F (Freeze Thread)**
 - **~U (Unfreeze Thread)**

When performing user-mode debugging, processes and threads can be controlled in various ways.

The *current* or *active* process is the process currently being debugged. Similarly, the *current* or *active* thread is the thread that is being controlled by the debugger. These control the actions of many debugger commands, as well as determining the virtual address mappings used by the debugger.

When debugging begins, the current process is the one that the debugger is attached to or that caused the exception that broke into the debugger. Likewise, the current thread is the one that was active when the debugger attached to the process, or that caused the exception. However, you can use the debugger to change the current process and thread, and to freeze or unfreeze individual threads.

Unless you enabled the debugging of child processes when starting the debugging session, there will only be one process available to the debugger.

To display process and thread information, the following methods are available:

- The **| (Process Status)** command
- The **~ (Thread Status)** command
- (*WinDbg only*) The **Debug | Processes and Threads** menu command

To change the current process or thread, the following methods are available:

- The **|S (Set Current Process)** command
- The **~S (Set Current Thread)** command
- (*WinDbg only*) The **Debug | Processes and Threads** menu command

To freeze and unfreeze individual threads, the following methods are available:

- The **~F (Freeze Thread)** command
- The **~U (Unfreeze Thread)** command

There are also a large number of other commands which can be prefixed by thread specifiers or process specifiers. See the debugger documentation for details.

Controlling Exceptions and Events

- **Using the Debugger to Analyze an Exception**
- **Controlling Exceptions and Events from the Debugger**
- **Controlling Break Status**
- **Controlling Handling Status**
- **Automatic Commands**
- **Event Definitions and Defaults**

Exceptions in user-mode and kernel-mode programs can be caught and handled by a variety of methods. An active debugger, a postmortem debugger, or an internal error handling routine are all common ways to handle exceptions.

When Windows allows a debugger to handle an exception, the application that generated the exception will *break into the debugger*. This means that the application will halt and the debugger will become active. The debugger will then have the option of handling the exception in some way or otherwise analyzing the situation. The debugger can then terminate the process or let it resume running.

If the debugger ignores the exception and lets the program continue running, the operating system will look for other exception handlers as if no debugger was present. If the exception is handled, the program will continue running. However, if the exception remains unhandled the debugger will then be given a second chance to deal with the situation.

Case One: No Debugger is Present on the System

If Windows has not been notified of any debugger existing on the system, it will look for exception handlers in a fixed order. Exception handling within the code itself (such as a **try-except** mechanism) will get the first chance to handle the exception. If the exception is not handled in any way, the operating system will terminate the process.

Case Two: "Just-in-Time" Debugger is Present

If a debugger has been set up as the Just-in-Time Debugger but is not currently attached to the process, Windows will look for exception handlers in the standard order. If the exception is not handled by any of the standard mechanisms, the postmortem debugger will be started and will be given control of the process which threw the exception.

The debugger is then free to analyze the situation and either continue or terminate the process.

Case Three: Debugger is Already Attached to the Process

If an exception occurs in a process which is actively being debugged, Windows will contact debugger *before* looking for any other exception handler.

At this point, the application will *break into the debugger*—in other words, the application will halt and the debugger will become active. The debugger will then have the option of handling the exception in some way or otherwise analyzing the situation. The debugger can then terminate the process or let it resume running.

If the debugger ignores the exception and lets the program continue running, the operating system will look for other exception handlers as if no debugger was present. If the exception is handled, the program will continue running. However, if the exception remains unhandled the debugger will then be given a second chance to deal with the situation.

Using the Debugger to Analyze an Exception

When an exception or event breaks into the debugger, you can use it to examine the code being executed and the memory being used by the program. By altering certain quantities or jumping to a different point in the program, you may be able to remove the cause of the exception.

You can resume execution by issuing a **GH (Go with Exception Handled)** or **GN (Go with Exception Not Handled)** command.

If you issue the **GN** command in the face of a "second-chance" exception, the program will terminate.

Kernel-Mode Exceptions

Exceptions which occur in kernel-mode code are more serious than user-mode exceptions. If they are not handled, a bug check will be issued and the entire system will halt.

As with user-mode exceptions, if a kernel-mode debugger is attached to the system it will be notified before the bug check screen ("blue screen") appears. If no debugger is attached, the blue screen will appear. In this case, a crash dump file may be created.

Controlling Exceptions and Events from the Debugger

The debugger can be configured to react to specified exceptions and events in a pre-determined way.

The debugger can set the *break status* for each exception or event. The event can cause a break into the debugger as soon as it occurs (the "first chance"); it can break in after other error handlers have been given an opportunity to respond (the "second chance"); it can just send the debugger a message but continue executing; or it can be ignored entirely.

The debugger can also set the *handling status* for each exception and event. The event can be treated as a handled exception or as an unhandled exception. (Of course, events which are not actually errors will not require any handling.)

Break status and handling status can be controlled by the following methods:

- Use the **SXE**, **SXD**, **SXN**, or **SXI** command in the Debugger Command window.
- (*CDB only*) Use the **-x**, **-xe**, **-xd**, **-xn**, or **-xi** option on the CDB command line.
- (*CDB only*) Use the **sxe** or **sxd** keyword can be used in the *tools.ini* file.
- (*WinDbg only*) Use the **Debug | Event Filters** menu selection to open the **Event Filters** dialog box, and choose the desired options in this box.

The **SX*** command, the **-x*** command-line option, and the **sx* Tools.ini** keyword normally set the break status of the specified event. Adding the **-h** option causes the handling status to be set instead.

There are four special event codes (**cc**, **hc**, **bpec**, and **ssec**) which always specify handling status rather than break status.

You can display the most recent exception or event by using the **.lastevent (Display Last Event)** command.

Controlling Break Status

When you set the break status of an exception or event, the following options are possible:

Command	Status Name	Description
SXE or -xe	Break (Enabled)	When this exception occurs, the target will immediately break into the debugger. This will happen before any other error handlers are activated — this is called <i>first chance</i> handling.
SXD or -xd	Second chance break (Disabled)	The debugger will not break for a first-chance exception of this type (although a message will be displayed). If other error handlers then fail to address this exception, execution will halt and the target will break into the debugger. This is called <i>second chance</i> handling.
SXN or -xn	Output (Notify)	When this exception occurs, the target application will not break into the debugger at all. However, a message informing the user of this exception will be displayed.
SXI or -xi	Ignore	When this exception occurs, the target application will not break into the debugger at all, and no message will be displayed.

If an exception is not anticipated by an SX* setting, it will break into the debugger on the second chance. The default status for events is listed below.

To set break status using the WinDbg graphical interface, select **Debug | Event Filters** from the menu, click on the desired event from the list in the **Event Filters** dialog box, and then select **Enabled**, **Disabled**, **Output**, or **Ignore**.

Controlling Handling Status

All events will be considered unhandled unless the **GH (Go with Exception Handled)** command is used.

The same is true of exceptions, unless the **SX*** command is used with the **-h** option.

Additionally, there are three events (invalid handles, STATUS_BREAKPOINT break instructions, and single-step exceptions) which can have their handling status configured by the SX* options. This is separate from their break configuration. When configuring their break status, these events are respectively named **ch**, **bpe**, and **sse**. When configuring their handling status, they are named **hc**, **bpec**, and **ssec**. (See full listing of events below.)

The CTRL+C event (**cc**) can have its handling status configured, but not its break status. If an application receives a CTRL+C, it will always break into the debugger.

When using the **SX*** command on the **cc**, **hc**, **bpec**, and **ssec** events, or when using the **SX*** command with the **-h** option on an exception, the following effects result:

Command	Status Name	Description
SXE	Handled	The event will be considered handled when execution resumes.
SXD, SXN, SXI	Not Handled	The event will be considered not handled when execution resumes.

To set handling status using the WinDbg graphical interface, select **Debug | Event Filters** from the menu, click on the desired event from the list in the **Event Filters** dialog box, and then select **Handled** or **Not Handled**.

Automatic Commands

The debugger also allows you to set commands which will be automatically executed if the event or exception causes a break into the debugger. A command string for the first-chance break and a command string for the second-chance break are permitted. These can be set with the **SX*** command or the **Debug | Event Filters** menu command. Each command string can contain multiple commands separated with semicolons.

These commands will be executed regardless of the break status. In other words, if the break status is "Ignore," the command will still be executed; if the break status is "Second-chance break," the first-chance command will be executed when the exception first occurs, before any other exception handlers are involved. The command string can end with an execution command such as **G (Go)**, **GH (Go with Exception Handled)**, or **GN (Go with Exception Not Handled)**.

Event Definitions and Defaults

The following exceptions can have their break status or handling status changed. Their default break status is indicated.

Their default handling status is always Not Handled. You should be very careful about changing this — if you change this to Handled, all first-chance and second-chance exceptions of this type will be considered handled, and this will bypass all the exception-handling routines.

Event Code	Meaning	Default Break Status
av	Access violation	Break
dm	Data misaligned	Break
dz	Divide by zero	Break
eh	C++ EH exception	Second-chance break
gp	Guard page violation	Break
ii	Illegal instruction	Second-chance break
iov	Integer overflow	Break
ip	In-page I/O error	Break
isc	Invalid system call	Break
lsq	Invalid lock sequence	Break
sbo	Stack buffer overflow	Break
sov	Stack overflow	Break
wkd	Wake debugger	Break

aph	Application hang	Break
	This exception is triggered if Windows concludes that a process has "hung"	
3c	Child application termination	Second-chance break
ch	Invalid handle	Break
hc		
Number	Any numbered exception	Second-chance break

Note

The **ch** and **hc** event codes refer to the same exception. When you are controlling its break status, use **sx* ch**. When you are controlling its handling status, use **sx* hc**.

The following exceptions can have their break status or handling status changed. Their default break status is indicated.

Their default handling status is always Handled. Since these exceptions are used for communication with the debugger, you will usually not want to change their status to Not Handled, as this would cause other exception handlers to catch them if the debugger chooses to ignore them.

An application can use the **DBG_COMMAND_EXCEPTION** (**dbce**) to communicate with the debugger. This does not essentially differ from a breakpoint, but you can use the **SX*** command to react in a specific way when this occurs.

Event Code	Meaning	Default Break Status
dbce	Special debugger command exception	Ignore
vcpp	Special Visual C++ exception	Ignore
wos	WOW64 single-step exception	Break
wob	WOW64 breakpoint exception-	Break
sse	Single-step exception	Break
ssec		
bpe	Breakpoint exception	Break
bpec		
cce	CTRL+C or CTRL+BREAK	Break
cc	This exception is triggered if the target is a console application and CTRL+C or CTRL+BREAK is passed to it.	

Note

The final three exceptions in the table above have two different event codes each. When you are controlling their break status, use **sse**, **bpe**, and **cce**. When you are controlling their handling status, use **ssec**, **bpec**, and **cc**.

The following events can have their break status changed. Since these are not exceptions, their handling status is irrelevant.

Event Code	Meaning	Default Break Status
ser	System error	Ignore
cpr	Process creation This event can only be controlled if debugging of child processes has been activated through CDB's -o command-line option .	Ignore
epr	Process exit This event can only be controlled if debugging of child processes has been activated through CDB's -o command-line option .	Ignore
ct	Thread creation	Ignore
et	Thread exit	Ignore
Id[:Module]	Load module If <i>Module</i> is specified, the break will occur when the module with this name is loaded. If it is not specified, the break will occur when any module is loaded. The debugger will only remember the most recent Id setting; separate settings for separate modules are not supported. There should be either a colon or a space between Id and <i>Module</i> .	Output
ud[:Module]	Unload module If <i>Module</i> is specified, the break will occur when the module with this name or at this base address is unloaded. If it is not specified, the break will occur when any module is unloaded. The debugger will only remember the most recent ud setting; separate settings for separate modules are not supported. There should be either a colon or a space between ud and <i>baseaddr</i> .	Output
out	Target application output	Ignore
ibp	Initial break point (This occurs at the beginning of the debug session, as well as after restarting or rebooting.)	<i>In user mode:</i> Break. This can be changed to Ignore by using the -g command-line option. <i>In kernel mode:</i> Ignore. This can be changed to Enabled by a variety of methods.

iml	Initial module load (Kernel mode only)	Ignore. This can be changed to Break by a variety of methods.
------------	--	---

Debugging an Application Failure

- **Most common Kinds of Application Failures**
 - **Access Violations**
 - **Alignment Faults**
 - **Exceptions**
 - **Critical Section Timeouts (deadlocks)**
 - **In-Page I/O Errors**

There are a large variety of errors possible in user-mode applications.

The most common kinds of failures include access violations, alignment faults, exceptions, critical section timeouts (deadlocks), and in-page I/O errors.

Access violations and data type misalignments are among the most common. They usually occur when an invalid pointer is dereferenced. The blame could lie with the function which caused the fault, or with an earlier function which passed an invalid parameter to the faulting function.

User-mode exceptions have many possible causes. If an unknown exception occurs, look it up in *ntstatus.h* or *winerror.h* if possible.

Critical section timeouts (or possible deadlocks) occur when one thread is waiting for a critical section for a long time. These are difficult to debug and require an in-depth analysis of the stack trace.

In-page I/O errors are almost always hardware failures. You can double-check the status code in *ntstatus.h* to verify.

Ending the Debugging Session

- **Exiting NTSD**
 - Q (Quit)
- **Ending the Session (WinDbg User-Mode)**
 - Debug | Stop Debugging
- **Exiting WinDbg**
 - Q (Quit)
 - File | Exit

You can exit any debugger at any time during the debugging session. You can also end the debugging session without exiting WinDbg.

Ending the Session but not Exiting

When performing user-mode debugging in WinDbg, you can end the debugging session without exiting WinDbg. This is done by selecting the **Debug | Stop Debugging** menu command. This can also be done by pressing the SHIFT+F5 shortcut key or by using the following button from the toolbar:



You will be prompted to save the workspace for the current session, and then WinDbg will return to dormant mode. At this point, all starting options are available to you: you can begin debugging a running process, spawn a new process, attach to a target machine, open a crash dump, or connect to a remote debugging session.

Exiting WinDbg

You can exit WinDbg by using the **Q (Quit)** command, by selecting the **File | Exit** menu command, or by using the ALT+F4 shortcut key. This also terminates the application you were debugging.

On Windows XP and later versions of Windows, the **QD (Quit and Detach)** command will detach WinDbg from the target application, exit the debugger, and leave the target application running. On Windows NT and Windows 2000, this command will generate a warning message and have no effect. These operating systems do not allow a debugger to detach from its target application.

User-Mode Dump Files

Creating a User-Mode Dump File

- Dr. Watson (`drwtsn32.exe`)
 - Using Dr. Watson to Create a Dump File
 - `drwtsn32 -p ProcessID`
 - Installing Dr. Watson as the JIT Debugger
 - `drwtsn32 -i`
- UserDump (`userdump.exe`)
 - Using UserDump to Create a Dump File
 - `userdump [ProcessID] or [ProcessName]`

When an application error occurs, Windows can respond in several different ways, depending on the postmortem debugging settings. If these settings instruct a debugging tool to create a dump file, a user-mode memory dump file will be created.

There are two tools which can create this file:

- Dr. Watson (`drwtsn32.exe`)
- UserDump (`userdump.exe`)

Both of these tools will create the same kind of user-mode dump files.

In addition, the **.dump** (Create User-Mode Crash Dump) command can be used to create a dump file. This command can be issued when the target application is still running. After the dump file is created, the target can continue to run.

The default postmortem setting is for a dump file to be written by Dr. Watson.

Using Dr. Watson to Create a Dump File

The Dr. Watson for Windows program (`drwtsn32.exe`) is preinstalled in your system directory (typically `c:\winnt\system32`) when Windows is set up. The default options are set the first time Dr. Watson for Windows runs, which can be either when an application error occurs or when you run it from the command prompt or the **Run** dialog box.

To create a `.dmp` of a process using Dr. Watson, the following command line is generally used:

`drwtsn32 -p ProcessID`

By default, Dr. Watson is set up to save a memory dump file immediately upon the failure of a user-mode component. By default, this file is named `user.dmp` and is saved in the `%AllUsersProfile%\Documents\DrWatson` folder.

Installing Dr. Watson as the Postmortem Debugger

The default registry settings for Windows install Dr. Watson as the postmortem debugger. This means that when an application crashes, Dr. Watson will be activated and will save a memory dump file.

If these settings have been changed, you can reinstall Dr. Watson as the postmortem debugger by running Dr. Watson with the **-i** option:

```
drwtsn32 -i
```

Displaying a Process List

A list of currently running processes and process IDs can be displayed by specifying the **-p** command line parameter. If **-p** is specified as the first argument, any other arguments are ignored.

```
C:\>userdump -p

User Mode Process Dumper (Version 3.0)
Copyright (c) 1999 Microsoft Corp. All rights reserved.

      0 System Idle Process
      2 System
     20 smss.exe
     24 csrss.exe
     34 WINLOGON.EXE
     40 SERVICES.EXE
     43 LSASS.EXE
     69 SPOOLSS.EXE
    92 RPCSS.EXE
   96 PSTORES.EXE
 124 NDDEAGNT.EXE
 207 EXPLORER.EXE
 208 systray.exe
 200 internat.exe
```

Dumping Running Process(es) by PID or Process Name

Userdump.exe can dump either a single process identified by a process ID, or each process matching the given image binary file name.

```
userdump <ProcessSpec> [<TargetDumpFile>]
```

If the process argument is a valid decimal or 0x-prefixed hex number, then it is interpreted as a process ID.

```
C:\>userdump 117 c:\x.dmp
User Mode Process Dumper (Version 3.0)
Copyright (c) 1999 Microsoft Corp. All rights reserved.

Dumping process 117 (CMD.EXE) to
C:\x.dmp...
The process was dumped successfully.
```

Otherwise the process argument is interpreted as the name of an image binary. The image binary name should be the base name with no path specifiers, i.e., "x.exe." Legal characters for the image name include ASCII characters in the printable range, except these characters: \ / " ? : * < > | and space. Non-printable and extended characters are not supported. A check for legal names is not explicitly done, but attempts to dump processes whose names include illegal, extended, or double-byte characters will fail. If there is

more than one instance of the application running, all get dumped; userdump.exe appends the process id to the base part of the dump filename in this case.

```
C:\>userdump cmd.exe c:\cmd.dmp
User Mode Process Dumper (Version 3.0)
Copyright (c) 1999 Microsoft Corp. All rights reserved.

Dumping process 117 (CMD.EXE) to
C:\cmd117.dmp...
The process was dumped successfully.

Dumping process 119 (CMD.EXE) to
C:\cmd119.dmp...
The process was dumped successfully.
```

If a dump file name is not specified, the dump file will be placed in the current directory using a name based on the image binary file name.

```
C:\mydir>userdump 117
User Mode Process Dumper (Version 3.0)
Copyright (c) 1999 Microsoft Corp. All rights reserved.

Dumping process 117 (CMD.EXE) to
C:\mydir\cmd.dmp...
The process was dumped successfully.

C:\mydir>userdump cmd.exe
User Mode Process Dumper (Version 3.0)
Copyright (c) 1999 Microsoft Corp. All rights reserved.

Dumping process 117 (CMD.EXE) to
C:\mydir\cmd117.dmp...
The process was dumped successfully.

Dumping process 119 (CMD.EXE) to
C:\mydir\cmd119.dmp...
The process was dumped successfully.
```

Multiple Process Case

If -m is specified, then multiple processes are dumped as close to simultaneously as possible. This is useful for a set of related processes, when the dumps obtained need to reflect the interdependent state of the apps at a particular point in time. Userdump will attempt to freeze in parallel the threads in all target processes before it takes any dumps of any of them.

```
userdump -m [-k] <ProcessSpec> [<ProcessSpec>...] [-d
<TargetDumpPath>]
```

If a process spec is a valid decimal or 0x-prefixed number, then it is interpreted as a process ID. Otherwise it is interpreted as the name of an image binary. The image binary name should be the base name with no path specifiers, i.e., "x.exe." Legal characters for the image name include ASCII characters in the printable range, except these characters: \ / " ? : * <> | and space. Non-printable and extended characters are not supported. A check for legal names is not explicitly done, but attempts to dump processes whose names include illegal, extended, or double-byte characters will fail.

Note that any combination of process IDs and image binary file names may be specified. Userdump dumps each single process specified by ID, and all running instances of each process specified by image binary file name.

Dump files are created by default in the current directory, using names that are constructed from the base name of the process' image binary file, plus the process ID, plus the .dmp extension.

```
C:\mydir>userdump -m 117 explorer.exe 119
User Mode Process Dumper (Version 3.0)
Copyright (c) 1999 Microsoft Corp. All rights reserved.

Dumping process 117 (CMD.EXE) to
C:\mydir\cmd117.dmp...
The process was dumped successfully.

Dumping process 207 (EXPLORER.EXE) to
c:\mydir\explorer207.dmp...
The process was dumped successfully.

Dumping process 119 (CMD.EXE) to
C:\mydir\cmd119.dmp...
The process was dumped successfully.
```

The directory where the dumps are placed can be overridden by specifying -d and a valid Win32 path to an existing directory, as the last 2 arguments on the command line.

```
C:\mydir>userdump -m 117 explorer.exe 119 -d c:\bar
User Mode Process Dumper (Version 3.0)
Copyright (c) 1999 Microsoft Corp. All rights reserved.

Dumping process 117 (CMD.EXE) to
C:\bar\cmd117.dmp...
The process was dumped successfully.

Dumping process 207 (EXPLORER.EXE) to
c:\bar\explorer207.dmp...
The process was dumped successfully.

Dumping process 119 (CMD.EXE) to
C:\bar\cmd119.dmp...
The process was dumped successfully.
```

Analyzing a User-Mode Dump File

- Analyzing a User-Mode Dump File with WinDbg
 - Starting WinDbg
 - Analyzing the Dump File
 - Reading a Dr. Watson Log

User-mode memory dump files can be analyzed by WinDbg. These dump files can be debugged on a different Windows version and different processor than the one they were created on.

Starting WinDbg

To analyze a dump file, start WinDbg with the **-z** command-line option:

windbg -y SymbolPath -i ImagePath -z DumpFileName

The **-v** option (verbose mode) is also useful.

If WinDbg is already running and is in dormant mode, you can open a crash dump by selecting the **File | Open Crash Dump** menu command or pressing the CTRL+D shortcut key. When the **Open Crash Dump** dialog box appears, enter the full path and name of the crash dump file in the **File name** text box, or use the dialog box to select the proper path and file name. When the proper file has been chosen, press **Open**.

Dump files generally end with the extension *.dmp*. You can use network shares or Universal Naming Convention (UNC) file names for the memory dump file.

Analyzing the Dump File

Analysis of a dump file is similar to analysis of a live debugging session.

If you are looking at a user.dmp file created by Dr. Watson because the process was unable to handle some sort of exception, then you will want to look for the thread in the process handling the exception to give you clues as to what you are looking for.

If you are looking at a dump file that was created as a snap-shot of a running process that appears to be hung or spinning, then you may want to look for threads that are either waiting on or in a critical section.

Reading a Dr. Watson Log

When an application exception (or program error) occurs, Dr. Watson generates a log file (Drwtsn32.log). The log file will always start with the following line

Application exception occurred:

The next part of the log file always contains program error information. The error number listed corresponds to the error generated by the system.

```
App: fault.exe (pid=141) ] Program that
When: 6/16/1993 @ 15:24:48.15
Exception number: c0000005 (access violation) ] Error
                                            that occurred
```

The next part of the log file contains system information about the user and the computer on which the program error occurred.

```
*--> System Information <---*
Computer Name: WESWX86
User Name: wesw
Number of Processors: 1
Processor Type: Intel 486
Windows Version: 3.10 ] General system information
                                            about the computer on which
                                            the program error occurred.
```

The next part of the log file contains the list of tasks that were running on the system at the time that the program error occurred.

```
*--> Task List <---*
0 Idle.exe
7 System.exe
28 smss.exe
20 csrss.exe
13 winlogon.exe
69 screg.exe
64 lsass.exe
62 spoolss.exe
48 EventLog.exe
200 CMD.exe
141 fault.exe
185 drwtsn32.exe ] Task Name
                    ] Task Identifier
```

The next part of the log file contains the list of modules that the program loaded.

```
*--> Module List <---*
(00010000 - 00021000) fault.exe
(76e50000 - 76ea8000) C:\winnt\nt\system32\ntdll.dll
(77d50000 - 77da5000) C:\winnt\nt\system32\srtl1.dll
(77730000 - 777a0000) C:\winnt\nt\system32\kernel32.dll ] Module Name
                ] Starting Address   ] Ending Address
```

The next part of the log file contains the state dump for the thread ID that is listed. The state dump consists of a register dump, disassembly of the code surrounding the current program counter, a stack back trace, and a raw stack dump. This first part of the state dump lists the thread ID.

State Dump for Thread Id 0xbf

The next part of the state dump contains the register dump.

```
eax=00000000 ebx=7ffe0000 ecx=00011277 edx=00152360 esi=002e00c8 edi=002e005c
  Register Name      Register Value
```

The next part of the state dump contains the instruction disassembly.

```
Function Name
function: AccessViolation
    Raw Machine Instruction
    00011670 55
    00011671 8bec
Address 00011673 53
    00011674 56
    00011675 57
    00011676 2bc0
    FAULT ->00011678 c700000000000000
    push    ebp
    mov     esp,esp
    push    ebx
    push    esi
    push    edi
    sub     eax eax
    mov     dword ptr [eax],0x0      ds:00000000=??
  Faulting Instructions
```

The next part of the state dump contains the stack back trace.

```
*----> Stack Back Trace <----*
RetAddr FramePtr Param#1 Param#2 Param#3 Param#4
00011678 0014eb94 002e005c 002e00c8 7ffe0000 00000065 AccessViolation
76e53b44 0014ecf8 002e0060 002e005c 002e0000 7ffe0000 CsrpProcessCallbackRequest
  Frame Pointer   First 4 Parameters   Function Name
  Return Address
```

The final part of the state dump contains the raw stack dump.

```
*----> Raw Stack Dump <----*
0014ec98 0a 02 48 00 02 02 00 00 - 00 00 00 00 d8 00 24 00 ...H.....$.
0014eca8 5c 00 2e 00 60 00 2e 00 - 00 f0 fe 7f 02 02 00 00 \....'.
0014ecb8 c0 0c 15 00 f8 ec 14 00 - 76 48 4f 76 0a 02 48 00 .....vHOv..H.
  Address       Hexadecimal Data          ASCII Data
```

Following the state dump, the final part of the log file contains the symbol table (if enabled).

```
*----> Symbol Table <----*
Module Name [fault.exe]
    00010000 _except_list
    00011000 main
    00011277 MyWndProc
    00011484 GetCommandLineArgs
    00011670 AccessViolation
  Address           Function Name
```

Debugging Memory Leaks

- **User-Mode Memory Leaks**
 - Using Performance Monitor to Detect a Leak
 - Using UMDH to Find User-Mode Memory Leaks

User-Mode Memory Leaks

A memory leak is caused by an application or process that allocates memory for use but does not later free it up when done. The result is that available memory is completely used up over time, often causing the system to stop functioning properly.

Using Performance Monitor to Detect a Leak

Like finding kernel-mode leaks, you can check for major memory leaks using the Performance MMC utility.

To determine whether or not a process is experiencing memory leaks, use Windows Performance Monitor (Perfmon.exe) and monitor "Private Bytes" under the Process category for your application. Private Bytes is the total amount of memory the process has allocated but is not sharing with other processes. Note that this is different from "Virtual Bytes," which is also interesting to monitor. Virtual Bytes is the current size in bytes of the virtual address space the process is using. An application can leak virtual memory but not see a difference in the amount of private bytes allocated. If you don't see memory rising when monitoring private bytes but suspect that you are still running out of memory, you should monitor virtual bytes to see if you are exhausting virtual memory.

Using UMDH to Find User-Mode Memory Leaks

The user-mode dump heap (UMDH) utility works with the operating system to analyze Windows heap allocations for a specific process. This utility and the other tools associated with it are targeted for Windows 2000 and later NT-based operating systems.

A self-extracting executable is included that contains the following tools:

- **Umdh.exe-** This utility is used to dump the heap allocation information for a process.

- Gflags.exe This utility sets the appropriate registry entries for the application that will be analyzed. The operating system looks at the registry entries to determine if an application's heap allocations will be tracked.
- Tlist.exe - This application lists all of the processes running on a machine and all of their related process IDs.
- Dhcmp.exe - This tool is used to compare two UMDH dumps to determine where a possible memory leak is occurring.
- Dhcmpgui.zip - This tool also compares two UMDH logs but has a user interface to make it easier to retrieve information. The .zip file contains the Dhcmpgui.exe file and its source code.

Capturing Heap Dumps with UMDH

UMDH is a tool that dumps information about a process's heap allocations. This information includes the callstack for each allocation, the number of allocations made through that callstack, and the number of bytes consumed through that callstack.

For example:

```
00005320 bytes in 0x14 allocations (@ 0x00000428) by: BackTrace00053
ntdll!RtlDebugAllocateHeap+0x000000FD
ntdll!RtlAllocateHeapSlowly+0x0000005A
ntdll!RtlAllocateHeap+0x00000080
MyApp!_heap_alloc_base+0x00000069
MyApp!_heap_alloc_dbg+0x000001A2
MyApp!_nh_malloc_dbg+0x00000023
MyApp!_nh_malloc+0x00000016
MyApp!operator new+0x0000000E
MyApp!LeakyFunc+0x0000001E
MyApp!main+0x0000002C
MyApp!mainCRTStartup+0x000000FC
KERNEL32!BaseProcessStart+0x0000003D
```

This UMDH output shows that there were 21280 (0x5320) bytes allocated total from the callstack. The 21280 bytes were allocated from 20 (0x14) separate allocations of 1064 bytes (0x428). The callstack is given a identifier of BackTrace00053.

To produce a dump of the heap allocations, you must first let the operating system know that you would like the kernel to track the allocations. This is done by using the Gflags.exe utility.

Let's assume that you want to dump the heap contents for Notepad.exe. By the way, most processes have more than one heap active and the UMDH log will contain all of them. First you need to enable stack trace acquisition for the application you want to test. This feature is not enabled by default. The command to enable it is:

```
gflags -i notepad.exe +ust
```

The command does not enable stack tracing for processes that are already running, but all future executions of Notepad.exe will have it enabled. Alternatively, you can set the flag through the GFLAGS user interface (run Gflags.exe without any arguments to get the user interface). Before using UMDH, you also need to install the correct debug symbols for the components of your application as well as the operating system. Typically, the operating system symbols are installed in the SYMBOLS folder on the WINNT folder. UMDH attempts to find the symbol files by using the _NT_SYMBOL_PATH. The command for setting the path from a command prompt is:

```
set _NT_SYMBOL_PATH=%windir%\symbols
```

With the image flags set and the symbols installed, you are ready to start Notepad. After the program is started, you need to determine the Process ID (PID) of the Notepad Process just started. The command for this is:

```
tlist
```

You can find the PID from the output of the TLIST application. The PID information can also be obtained from Task Manager. Let's assume the PID for the Notepad process you just started is 124. You can use UMDH to get a heap dump by using the following command:

```
umdh -p:124 -f:notepad124.log
```

Results: You've got a complete heap dump of the Notepad process in the Notepad124.log file.

Using Dhcmp.exe to Compare UMDH Logs

While the UMDH log file contains valuable information about the current state of the heaps for a process, if you are concerned with finding a memory leak, it may be more valuable to compare the outputs of two logs and find out what callstack has seen the largest growth between the two dumps. The Dhcmp.exe utility helps compare two UMDH logs to provide an analysis of the difference between them.

Once you have two logs captured at different intervals, you can then use the following command:

```
DHCMPP dh1.log dh2.log > cmp12.txt
```

The .txt files will compare the effect on memory of running the suspect memory hog application. The output of the file generated by DHCMPP resembles the following:

```
+ 5320 ( f110 - 9df0) 3a allocs BackTrace00053  
Total increase == 5320
```

For each BackTrace in the UMDH log files, there is a comparison made between the two logs files. This case shows that the last log file specified in the DHCMPP command line had F110 bytes allocated while the first log in the DHCMPP command line had 9DF0 bytes allocated for the same BackTrace (callstack). The 5320 is the difference in the number of bytes allocated. In this case, there were 5320 more bytes allocated between the time the two logs were captured. The bytes came from the callstack identified by BackTrace00053.

The next step is to find out what's in that backtrace. If you open the second DH log file and search for "BackTrace00053" you might find something that resembles the following:

```
00005320 bytes in 0x14 allocations (@ 0x00000428) by: BackTrace00053
ntdll!RtlDebugAllocateHeap+0x000000FD
ntdll!RtlAllocateHeapSlowly+0x0000005A
ntdll!RtlAllocateHeap+0x00000808
MyApp!_heap_alloc_base+0x00000069
MyApp!_heap_alloc_dbg+0x000001A2
MyApp!_nh_malloc_dbg+0x00000023
MyApp!_nh_malloc+0x00000016
MyApp!operator new+0x0000000E
MyApp!LeakyFunc+0x0000001E
MyApp!main+0x0000002C
MyApp!mainCRTStartup+0x000000FC
KERNEL32!BaseProcessStart+0x0000003D
```

By looking at the callstack, you can see that the LeakyFunc function is allocating memory by way of the Visual C++ run-time library. If you find that the number of allocations grows as you took more dumps, you might be able to conclude that memory is not being freed.

Enabling Stack Traces

The most important information in UMDH logs is the stack traces of the heap allocations. By analyzing them it can be understood whether a process leaks heap memory. These stack traces are not acquired by default. The feature can be enabled per-process or system-wide. Use the following command to enable stack tracing system-wide:

```
gflags -r +ust
```

Restart the computer after this command.

For per-process enabling, the command is

```
gflags -i APPNAME +ust
```

where APPNAME is the file name of the executable including the extension (Services.exe, lsass.exe, and so on). The command will not enable stack tracing for a process that is already running. For this reason, in the case of processes that can't be restarted (services, lsass, winlogon), the test compute must be restarted.

Use the following command to verify what settings have been set system-wide:

```
gflags -r
```

Use the following command to verify what settings have been set for a specific process:

```
gflags -i APP-NAME
```

By default, the maximum stack trace depth is 16. If you want to see deeper callstacks, you can increase this by running GFLAGS. Click to select System Registry and type a new depth in the Max. Stack Trace Capture Depth edit control. Click Apply, and then restart the computer.

Invoking UMDH

The only required command-line parameter for UMDH is the -p option, which specifies the PID of the process from which a heap dump will be obtained. The PID can be obtained by using Task Manager or the Tlist.exe program. For a command similar to the following, the log will be dumped to the standard output:

```
umdh -p:PID
```

UMDH also displays various informational messages to standard error, and therefore if not redirected it will be mixed with the real log. To gather the UMDH informational messages in a file, use the following command:

```
umdh -p:PID 2>umdh.msg
```

If you want to gather the log dumped by UMDH in a file, use one of the following commands:

```
umdh -p:PID >umdh.log
```

or

```
umdh -p:PID -f:umdh.log
```

The commands are completely equivalent.

The default log obtained by UMDH contains an enumeration of heap consumers sorted by allocation count. If, for debugging purposes, you also need a dump of all allocated blocks with their corresponding stack traces, the -d option must be used:

```
umdh -p:PID -d
```

If the log contains too much information, it can be limited only to big users that have the allocation count above a certain threshold. Use the following command:

```
umdh -p:PID -t:THRESHOLD
```

All the command-line options (-p, -f, -t, -d) can be specified simultaneously in any order. Following is a complicated command-line example:

```
umdh -p:123 -t:1000 -f:umdh.log -d
```

This command dumps the heaps for the process with PID 123 into the Umdh.log file. It dumps only stack traces that account for more than 1000 allocations and it also dumps the addresses of the heap blocks allocated through each stack trace.

UMDH Output Explained

If you redirected the log to a file ("umdh -p:PID -f:umdh.log"), the contents will resemble the following, which was obtained from a running Notepad process:

```

UMDH: Logtime 2000-06-28 10:54 - Machine=MYMachine - PID=704
***** Heap 00270000 Information *****
Flags: 58000062
Number Of Entries: 87
Number Of Tags: <unknown>
Bytes Allocated: 00008DF0
Bytes Committed: 0000A000
Total FreeSpace: 00001210
Number of Virtual Address chunks used: 1
Address Space Used: <unknown>
Entry Overhead: 8
Creator: (Backtrace00007)
ntdll!RtlDebugCreateHeap+0x000000196
ntdll!RtlCreateHeap+0x0000023F
ntdll!LdrpInitializeProcess+0x00000369
ntdll!LdrpInitialize+0x0000028D
ntdll!KiUserApcDispatcher+0x00000007
***** Heap 00270000 Hogs *****
000001A0 bytes in 0x4 allocations (@ 0x00000068) by: BackTrace00031
ntdll!RtlDebugAllocateHeap+0x000000FB
ntdll!RtlAllocateHeapSlowly+0x0000005B
ntdll!RtlAllocateHeap+0x00000D81
ntdll!LdrpAllocateDataTableEntry+0x00000039
ntdll!LdrpMapDll+0x000002A4
ntdll!LdrpLoadImportModule+0x0000010D
ntdll!LdrpWalkImportDescriptor+0x0000008B
ntdll!LdrpLoadImportModule+0x00000011D
ntdll!LdrpWalkImportDescriptor+0x0000008B
ntdll!LdrpLoadImportModule+0x00000011D
ntdll!LdrpWalkImportDescriptor+0x0000008B
ntdll!LdrpInitializeProcess+0x000009DC
ntdll!LdrpInitialize+0x0000028D
ntdll!KiUserApcDispatcher+0x00000007

000001A0 bytes in 0x4 allocations (@ 0x00000068) by: BackTrace00034
ntdll!RtlDebugAllocateHeap+0x000000FB
ntdll!RtlAllocateHeapSlowly+0x0000005B
ntdll!RtlAllocateHeap+0x00000D81
ntdll!LdrpAllocateDataTableEntry+0x00000039
ntdll!LdrpMapDll+0x000002A4
ntdll!LdrpLoadImportModule+0x0000010D
ntdll!LdrpWalkImportDescriptor+0x0000008B
ntdll!LdrpLoadImportModule+0x00000011D
ntdll!LdrpWalkImportDescriptor+0x0000008B
ntdll!LdrpLoadImportModule+0x00000011D
ntdll!LdrpWalkImportDescriptor+0x0000008B
ntdll!LdrpLoadImportModule+0x00000011D
ntdll!LdrpWalkImportDescriptor+0x0000008B
ntdll!LdrpInitializeProcess+0x000009DC
ntdll!LdrpInitialize+0x0000028D
ntdll!KiUserApcDispatcher+0x00000007

```

The log contains a dump of every heap in the process. In this example, the log starts with a heap at address 270000. After a few global counters for the heap, the log contains a dump in decreasing sorted order of stack traces that are responsible for the most allocations. By comparing the dynamics of memory used at different moments you can deduce what happened in the process and if any heap use looks like a leak.

Debugging User-Mode System Processes

Redirecting NTSD Output to KD

- Allows User-Mode Debugging to be Performed Early in the Boot Phase
- Allows User-Mode Debugging to be Performed Fairly Late during Shutdown
- `ntsd -d appname`

The output from NTSD can be redirected to the kernel-mode debugger KD. This allows user-mode debugging to be performed through the serial port very early in the boot phase and fairly late into shutdown, since NTSD will be running on the host computer, rather than on the same computer as the application being debugged.

Use the `-d` command-line option to redirect NTSD's output to KD:

`NTSD -d appname`

All NTSD output is sent to the kernel debugger through the **DbgPrint** buffer. The NTSD command prompt appears on your debug machine and the target machine is halted until you continue. This is similar to standard kernel debugging, except that you are in a process context rather than the system context.

Another use of this technique is to begin analysis in user mode, and then jump into the kernel debugger in the context of the user-mode process.

When NTSD is already running, use the `!userexts.kbp` extension at the NTSD prompt. This breaks into KD.

To return to NTSD, type **G (Go)** at the KD prompt.

Currently, the combination of NTSD and KD in this manner provides the heavy-hitting, low-level debug analysis that results in resolution of the most difficult system and application bugs.

Debugging CSRSS with NTSD

- CSRSS is the executable that controls the underlying layer for the Windows® environment
- Using NTSD (why)
- Enabling CSRSS Debugging (gflags)
- Starting NTSD
 - ntsd --
 - ntsd -p -1 -g -G (The process ID of CSRSS is always -1.)

CSRSS is the executable that controls the underlying layer for the Windows® environment. There are a number of problems which make it necessary to debug CSRSS itself.

Debugging CSRSS is also useful when the Windows subsystem terminates unexpectedly with a Bug Check 0xC000021A: STATUS_SYSTEM_PROCESS_TERMINATED. In this case, debugging CSRSS will catch the failure before it gets to the "unexpected" point.

Although CSRSS is an integral part of the Windows operating system, it is actually a user-mode process.

Using NTSD

Normally, user-mode processes can be debugged equally well by CDB or NTSD. However, it is usually necessary to use NTSD to debug CSRSS. The reason for this is that NTSD is capable of running without any console window at all. If the CSRSS process is having problems or is frozen, a console application like CDB will not work properly.

Enabling CSRSS Debugging

CSRSS debugging must be enabled before you can proceed. If the target computer is running a checked build, CSRSS debugging is always enabled. If the target computer is running a free build, you will have to manually clear the CSRSS-debugging-disable bit in the registry of the target computer.

Use *gflags* or *regedit32* to edit the following registry key:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager

Edit the **GlobalFlag** value entry (of type REG_DWORD) and set the bit 0x00020000.

The registry change requires a reboot to take effect.

Starting NTSD

After the registry has been properly configured, it is a simple matter of starting NTSD as follows:

ntsd --

This command is simply shorthand for the following:

ntsd -p -1 -g -G

(The process ID of CSRSS is always -1.)

To break into NTSD by hand while debugging CSRSS, press F12.

Note that you may see an "in page io error" message. This is another manifestation of a hardware failure.

Debugging WinLogon with NTSD

- WinLogon is the executable which handles interactive user logons and logoffs, and handles all instances of CTRL+ALT+DELETE.
- Using NTSD and KD (why)
- Enabling WinLogon Debugging (Image File Execution Options)
- Gflags
 - Enable heap tail checking
 - Enable heap validation on call

WinLogon is the executable which handles interactive user logons and logoffs, and handles all instances of CTRL+ALT+DELETE.

Although WinLogon is an integral part of the Windows operating system, it is actually a user-mode process.

Using NTSD and KD

Normally, user-mode processes can be debugged equally well by CDB or NTSD. However, it is usually necessary to use an NTSD-KD combination to debug WinLogon. The NTSD debugger will run on the computer being debugged, and will send its output to KD on a second computer.

The reason for using NTSD rather than CDB is that NTSD is capable of running without any console window at all. If the WinLogon process is having problems or is frozen, a console application like CDB will not work properly.

Enabling WinLogon Debugging

To attach a debugger to WinLogon, you must go through the registry so that the process is debugged from the time it starts up. To set up WinLogon debugging, set

**HKEY_LOCAL_MACHINE\Software\Microsoft\Windows
NT\CurrentVersion\Image File Execution Options\WinLogon.EXE\Debugger** to:

ntsd -d -g

The **-d** option sends all NTSD output to KD, and **-g** causes it to go initially). Don't add the **-g** if you want to start debugging before *winlogon.exe* begins – for example, if you want to setup any initial breakpoints. Pressing F12 on the target computer will break into *winlogon.exe* and give you an NTSD debug prompt on the debug computer. In addition to the debugger entry, you should set the GLOBAL_FLAG value under the *winlogon.exe* key to REG_SZ "0x000400f0" (Enable heap tail checking and Enable heap validation on call). These changes will take effect on the next reboot.

Module 11 Labs

