

Loading EXE and DLL Programs

Matt Pietrek

Note: Several of the figures mentioned are located at the end of this article.

I've discussed COM type libraries and database access layers such as ActiveX® Data Objects (ADO) and OLE DB. Longtime readers of my MSJ writings (both of them) probably think I've gone soft. To redeem myself, this month I'll tour part of the Windows NT® loader code where the operating system and your code come together. I'll also demonstrate a nifty trick for getting loader status information from the loader, and a related trick you can use in the Developer Studio® debugger.

Consider what you know about EXEs, DLLs, and how they're loaded and initialized. You probably know that when a C++ DLL is loaded, its DllMain function is called. Think about what happens when your EXE implicitly links to some set of DLLs (for example, KERNEL32.DLL and USER32.DLL). In what order will those DLLs be initialized? Is it possible for one of your DLLs to be initialized before another DLL that you depend on? The Platform SDK has this to say under the "Dynamic-Link Library Entry-Point Function" section.

Your function should perform only simple initialization tasks, such as setting up thread local storage (TLS), creating synchronization objects, and opening files. It must not call the LoadLibrary function, because this may create dependency loops in the DLL load order. This can result in a DLL being used before the system has executed its initialization code. Similarly, you must not call the FreeLibrary function in the entry-point function, because this can result in a DLL being used after the system has executed its termination code.

Calling Win32® functions other than TLS, synchronization, and file functions may also result in problems that are difficult to diagnose. For example, calling User, Shell, and COM functions can cause access violation errors, because some functions in their DLLs call LoadLibrary to load other system components.

Something I've learned firsthand is that the above documentation is still way too vague. For example, reading a registry key is a natural thing you'd want to do inside your DllMain function. It certainly qualifies as initialization. Unfortunately, in the right circumstances ADVAPI32.DLL isn't initialized before your DllMain code, and the registry APIs will just fail.

Given the stern warning about using LoadLibrary in the documentation, it's especially interesting that the Windows NT USER32.DLL explicitly ignores the preceding advice. You may be aware of a Windows NT only registry key called Applnit_Dlls that loads a list of DLLs into each process. It turns out that the actual loading of these DLLs occurs as part of USER32's initialization. USER32 looks at this registry key and calls LoadLibrary for these DLLs in its DllMain code. A little thought here reveals that the Applnit_Dlls trick doesn't work if your app doesn't use USER32.DLL. But I digress.

My point in bringing this up is that DLL loading and initialization is still a gray area. In most cases, a simplified view of how the OS loader works is sufficient. In those oddball 5 percent of cases, however, you can go nuts unless you have a more detailed working model of how the OS loader behaves.

Load 'er Up!

What most programmers think of as module loading is actually two distinct steps. Step one is to map the EXE or DLL into memory. As this occurs, the loader looks at the Import Address Table (IAT) of the module and determines whether the module depends on additional DLLs. If the DLLs aren't already loaded in that process, the loader maps them in as well. This procedure recurses until all of the dependent modules have been mapped into memory. A great way to see all the implicitly dependent DLLs for a given executable is the DEPENDS program from the Platform SDK.

Loading EXE and DLL Programs

Matt Pietrek

Step two of module loading is to initialize all of the DLLs. Stop and ponder this. While the OS loader is mapping the EXE and/or DLLs into memory in step one, it's not calling the initialization routines. The initialization routines are called after all the modules have been mapped into memory. Key point: the order in which DLLs are mapped into memory is not necessarily the same as the order in which the DLLs are initialized. I've seen people look at the DLL mapping notifications as they appear in the Developer Studio debugger and mistakenly assume that the DLLs were initialized in that same order.

In Windows NT, the routine that invokes the entry point of EXEs and DLLs is called `LdrpRunInitializeRoutines`, and it's worth taking a look at here. In my own work, I've stepped through the assembler code for `LdrpRunInitializeRoutines` many times. However, looking at a ream of assembler code isn't the best way to understand it. Therefore, I rewrote `LdrpRunInitializeRoutines` from Windows NT 4.0 SP3 in C++-like pseudocode, with the results shown in Figure 1. To be completely accurate, in `NTDLL.DBG` the routine name is `__stdcall` mangled to `_LdrpRunInitializeRoutines@4`. Also, in my pseudocode, unless a variable or structure name is prefixed with an underscore, it was a name I made up.

`LdrpRunInitializeRoutines` is the final stop in the Windows NT loader code before an EXE's or DLL's specified entry point is called. (In the following discussion, I'll use "entry point" and "initialization routine" interchangeably.) This loader code executes in the process context that loaded the DLL—that is, it's not part of some special loader process. `LdrpRunInitializeRoutines` is called at least once during process startup to handle implicitly loaded DLLs. `LdrpRunInitializeRoutines` is also called every time one or more DLLs is dynamically loaded, usually because of a call to `LoadLibrary`.

Each time `LdrpRunInitializeRoutines` executes, it seeks out and calls the entry point of all DLLs that have been mapped into memory, but not yet initialized. In examining the pseudocode, take note of all the extra code that provides trace output, even in the nonchecked builds of Windows NT. I'm referring to all the code that uses the `_ShowSnaps` variable and the `_DbgPrint` function. I'll come back to these players later.

At a high level, the function breaks up into four distinct sections. The first portion of the code calls `_LdrpClearLoadInProgress`. This `NTDLL` function returns the number of DLLs that have just been mapped into memory. For example, if you called `LoadLibrary` on `FOO.DLL` and `FOO` had implicit links to `BAR.DLL` and `BAZ.DLL`, `_LdrpClearLoadInProgress` would return 3 since three DLLs were mapped into memory.

After the number of DLLs to be concerned with is known, `LdrpRunInitializeRoutines` calls `_RtlAllocateHeap` (also known as `HeapAlloc`) to get memory for an array of pointers. In the pseudocode I've called this array `plnitNodeArray`. Each pointer in `plnitNodeArray` will eventually point to a structure containing information about the newly loaded (but not yet initialized) DLL.

In the second part of `LdrpRunInitializeRoutines`, the code digs into internal process data structures to obtain a linked list containing each of the newly loaded DLLs. As the code iterates through the linked list, it checks to see if the loader has somehow seen this DLL before (not likely). It also checks to ensure that the DLL has an entry point. If both tests are passed, the code appends the module information pointer to the `plnitNodeArray`. The pseudocode refers to the module information as `pModuleLoaderInfo`. Note that it's entirely possible for a DLL to not have an entry point—for example, a resource-only DLL. Thus, the number of entries in `plnitNodeArray` may be fewer than the value returned earlier by `_LdrpClearLoadInProgress`.

The third (and largest) section of `LdrpRunInitializeRoutines` is where things really start to happen. The code's mission here is to enumerate through each element in `plnitNodeArray` and call the entry point.

Loading EXE and DLL Programs

Matt Pietrek

Because of the very real possibility that a DLL's initialization code may fault, the entire third section of code is surrounded by a `__try` block. This is why a dynamically loaded DLL can fault in its `DllMain` without bringing the whole process down.

Iterating through an array and calling an entry point for each node should be a small task. However, some relatively obscure features of Windows NT add to the complexity. For starters, consider whether the process is being debugged by a Win32 debugger such as `MSDEV.EXE`. Windows NT has an option that allows you to suspend a process and send control to the debugger before a DLL is initialized. This feature is on a per-DLL basis, and is enabled by adding a string value (`BreakOnDllLoad`) to a registry key with the name of the DLL (for instance, `FOO.DLL`). See the pseudocode comment above the call to `_LdrQueryImageFileExecutionOptions` in Figure 1 for more information.

Another bit of extra code that may execute before a DLL's entry point invocation is the TLS initialization. When you declare TLS variables using `__declspec(thread)`, the linker includes data that causes this condition to be triggered. Right before the DLL's entry point is called, `LdrpRunInitializeRoutines` checks to see if a TLS initialization is necessary and, if so, calls `_LdrpCallTlsInitializers`. More on this later.

The moment of truth finally comes when `LdrpRunInitializeRoutines` calls the DLL's entry point. I deliberately left this part of the pseudocode in assembly language. You'll see why later. The crucial instruction is `CALL EDI`. Here, `EDI` points to the DLL's entry point, which is specified in the DLL's PE header. When `CALL EDI` returns, the DLL in question has completed its initialization. For DLLs written in C++, this means that the `DllMain` code has executed its `DLL_PROCESS_ATTACH` code. Also, note the third parameter to the entry point, normally referred to as `pvReserved`. In truth, this parameter is nonzero for DLLs that the EXE implicitly links to directly or through another DLL. The third parameter is zero for all other DLLs (that is, DLLs loaded as a result of a `LoadLibrary` call).

After the DLL entry point is invoked, `LdrpRunInitializeRoutines` does a sanity check to make sure the DLL entry point code was defined properly. The loader code looks at the stack pointer (`ESP`) value from before and after the entry point call. If they're different, something's wrong with the DLL's initialization function. Since most programmers never define the real DLL entry point function, this scenario rarely happens. However, when it does, you're informed of the problem via an onerous dialog (see Figure 2). I had to use a debugger and modify a register value at just the right spot to produce this dialog.

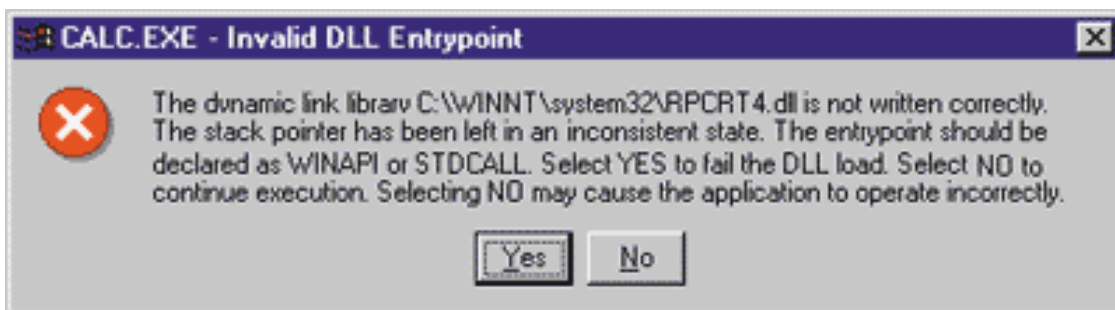


Figure 2 An Invalid DLL Entry Point

Following the stack check, `LdrpRunInitializeRoutines` checks the return code from the entry point routine. For C++ DLLs, this is the value returned from `DllMain`. If the DLL returned 0, it usually means something is wrong and that the DLL doesn't want to remain loaded. When this happens, you get the dreaded "DLL Initialization Failed" dialog.

Loading EXE and DLL Programs

Matt Pietrek

The final portion of the third section of `LdrpRunInitializeRoutines` occurs after all the DLLs have been initialized. If the process EXE itself has TLS data, and if the implicitly linked DLLs are being initialized, the code calls `_LdrpCallTlsInitializers`.

The fourth (and final) section of `LdrpRunInitializeRoutines` is the cleanup code. Remember earlier, when `_RtlAllocateHeap` created the `plnitNodeArray`? This memory needs to be freed, which occurs inside a `__finally` block. Even if one of the DLLs faults in its initialization code, the `__try/__finally` code ensures that `_RtlFreeHeap` is called to free `plnitNodeArray`.

This ends our tour of `LdrpRunInitializeRoutines`, so let's now look at some side topics that the code presents.

Debugging Initialization Routines

Every once in a while I come across a problem where a DLL is faulting in its initialization code. Unfortunately, the fault could be from any one of several DLLs, and the operating system doesn't tell me which DLL is the culprit. In these circumstances you can get sneaky and use a debugger breakpoint to narrow down the problem.

Most debuggers blithely skip past the initialization of statically linked DLLs. They're focused on getting you to the first instruction or first line in your EXE. However, knowing what `LdrpRunInitializeRoutines` looks like, you can set a breakpoint on the `CALL EDI` instruction where execution goes to the DLL entry point. Once the breakpoint is set, each time a DLL is about to get its `DLL_PROCESS_ATTACH` notification you'll stop in `NTDLL` at the `CALL` instruction. Figure 3 shows what this looks like in the Visual C++® 6.0 IDE (MSDEV.EXE).

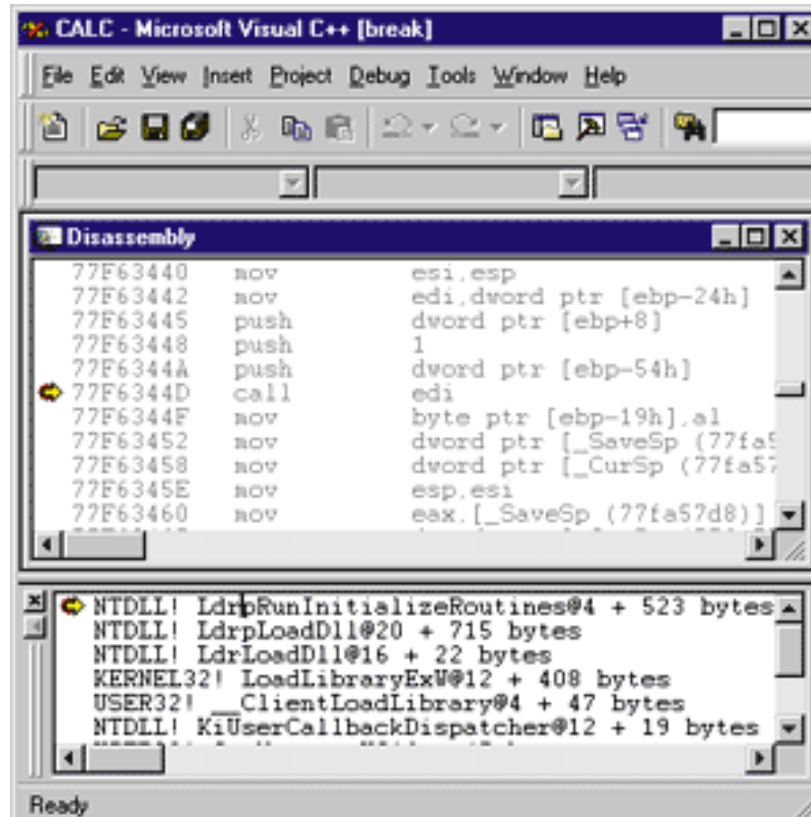


Figure 3 Setting a Breakpoint on CALL EDI

Loading EXE and DLL Programs

Matt Pietrek

If you choose to step into the CALL, you'll end up at the first instruction of the entry point of the DLL. It's important to understand that this code is almost never code you write. Rather, it's usually code in the runtime library that does its setup work and then calls your initialization code. For example, in a DLL written in Visual C++, the entry point is `_DllMainCRTStartup`, which is in `CRTDLL.C`. Without symbol tables or source code, what you'll see in the MSDEV assembly window will look something like Figure 4.

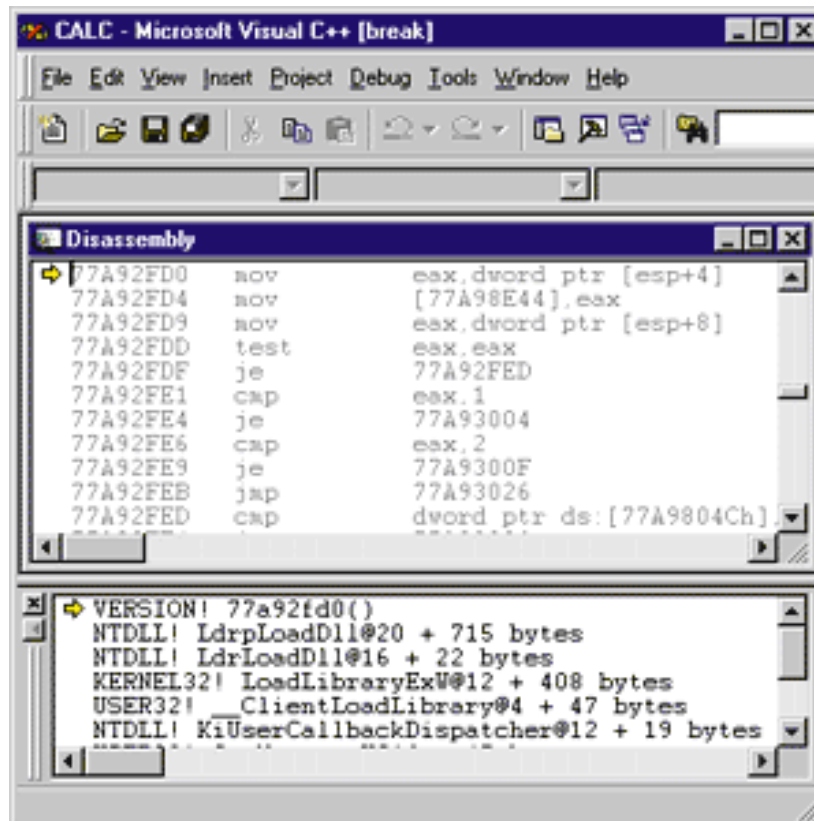


Figure 4 Stepping into the CALL

Usually my debugging process follows a predictable pattern. Step one is to figure out which DLL is faulting. Do this by setting the aforementioned breakpoint, and make one instruction step into each DLL as it initializes. Using the debugger, figure out which DLL you're in, and write it down. One way to do this is to use the memory window to look on the stack (ESP) and obtain the HMODULE of the DLL you've entered.

After you know which DLL you've entered, let the process continue (Go). In short order, the breakpoint should be hit again for the next DLL. Repeat this as often as necessary until you identify the problem DLL. You'll recognize the problem DLL because it will be called to initialize, but the process terminates before the initialization code returns.

Step two is to drill into the faulting DLL. If the offending DLL is one that you have source for, try setting a breakpoint on your `DllMain` code, then let the process run to see if your breakpoint is hit. If you don't have source, just run the process. Your breakpoint on the CALL EDI instruction should still be in place from before. Keep running until you get to the one where the initialization faults. Step into this entry point, and keep stepping until you can ascertain the problem. This may require stepping through a lot

Loading EXE and DLL Programs

Matt Pietrek

of assembly code! I never said this was easy, but sometimes it's the only way to hunt the problem down.

Finding the CALL EDI instruction can be tricky (at least with the current Microsoft® debuggers). You can see why I deliberately left this part of the pseudocode in assembler. For starters, you'll definitely need to have the correct NTDLL.DBG in your SYSTEM32 directory, alongside NTDLL.DLL. The debugger should automatically load the symbol table when you begin stepping through your program.

Using the assembly window in Visual C++, you can (in theory) goto an address using a symbolic name. Here, you'd want to go to `_LdrpRunInitializeRoutines@4` and then scroll down until you see the CALL EDI instruction. Unfortunately, the Visual C++ debugger doesn't recognize NTDLL symbol names unless you're already stopped in NTDLL.DLL.

If you happen to know the address of `_LdrpRunInitializeRoutines@4` (for instance, 0x77F63242 in Windows NT 4.0 SP 3 for Intel), you can type that in and the assembly window will happily display it. Heck, the IDE will even show you that it's the start of a function called `_LdrpRunInitializeRoutines@4`. If you're not a debugger guru, the failure to recognize the symbol name is extremely confusing. If you are a debugger nut like me, it's extremely annoying because you know what's causing the problem.

WinDBG from the Platform SDK is a little better about recognizing symbol names. Once you've started the target process, you can set a breakpoint on `_LdrpRunInitializeRoutines@4` using its name. Unfortunately, the first time you execute the process, execution blows past `_LdrpRunInitializeRoutines@4` before you get a chance to set your breakpoint. To remedy this, start WinDBG, make one instruction step, set the breakpoint, stop debugging, and remain in the debugger. You can then restart the debuggee and the breakpoint will be hit on every invocation of `_LdrpRunInitializeRoutines@4`. This same trick works in the Visual C++ debugger.

What's This ShowSnaps Thing?

One of the first things that jumped out at me when I looked at the `LdrpRunInitializeRoutines` code was the `_ShowSnaps` global variable. Now's a good time to briefly divert to the subject of the `GlobalFlag` and `GFlags.EXE`.

The Windows NT registry contains a DWORD value that influences certain behaviors of system code. Most of these modifications are heap- and debugging-related. The `GlobalFlag` value of the registry key

`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager`

is a set of bitfields. Knowledge Base article Q147314 describes most of the bitfields, so I won't go into them here. In addition to this systemwide `GlobalFlag` value, individual executables can have their own distinct `GlobalFlag` value. The process-specific `GlobalFlag` value is found under

`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT
\CurrentVersion\Image File Execution Options\imagename`

where `imagename` is the name of an executable (for instance, `WinWord.exe`). All of these documentation-challenged bitfields and highly nested resource keys scream out for a program to simplify it all. Wouldn't you know it, Microsoft has just such a program.

Loading EXE and DLL Programs

Matt Pietrek

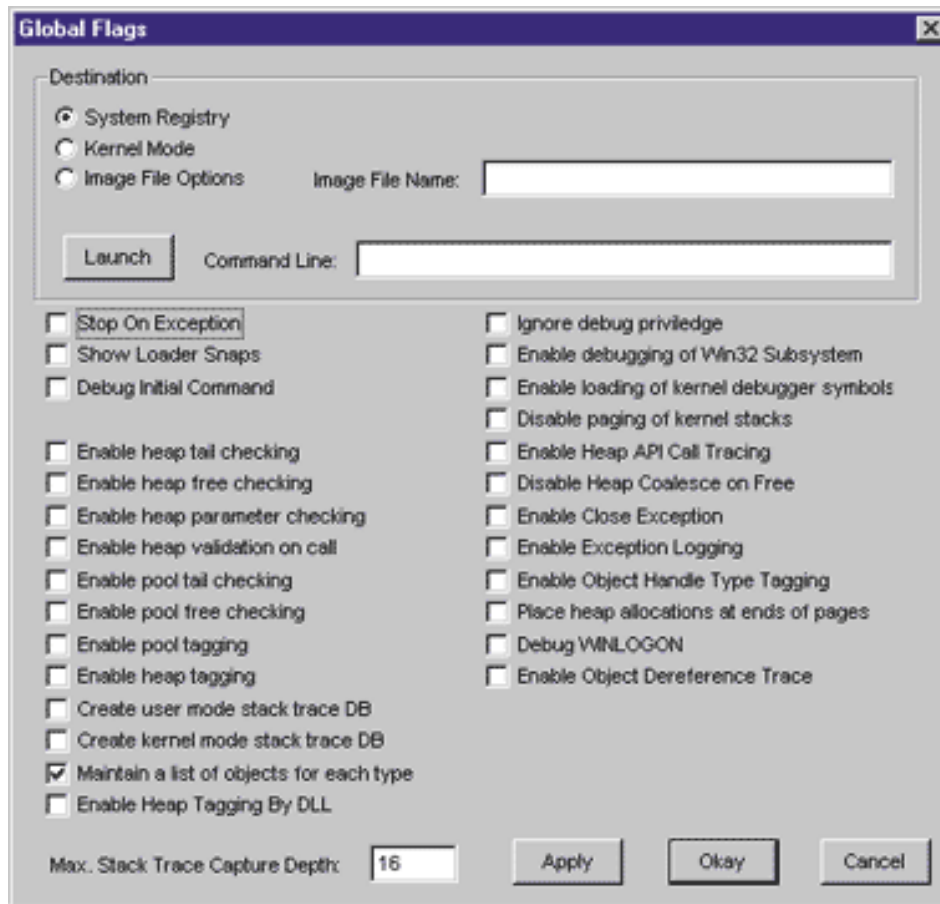


Figure 5 GFlags.EXE

Figure 5 shows GFlags.EXE, which comes in the Windows NT 4.0 Resource Kit. Near the top-left of the GFlags window are three radio buttons. Selecting either of the top two (System Registry or Kernel Mode) lets you make changes to the global Session Manager value of GlobalFlags. If you select the third radio button (Image File Options), the set of available option checkboxes shrinks dramatically. This is because some of the GlobalFlag options only affect kernel mode code and don't make sense on a per-process basis. It's important to note that most of the kernel mode-only options assume you're using a system-level debugger such as i386kd. Without such a debugger to poke around or receive the output, there's not much use in enabling these options.

To tie this back to the subject of `_ShowSnaps`, enabling the Show Loader Snaps option in GFlags causes the `_ShowSnaps` variable to be set to a nonzero value in NTDLL.DLL. In the registry, this bit value is `0x00000002`, which is #defined as `FLG_SHOW_LDR_SNAPS`. Luckily, this bitflag is one of the GlobalFlag values that can be enabled on a per-process basis. The output can be quite voluminous if you enable it systemwide.

Examining ShowSnaps Output

Let's take a look at what sort of output enabling Show Loader Snaps produces. It turns out that other parts of the Windows NT loader that I haven't discussed also check this variable and emit additional output. Figure 6 shows some abbreviated output from running CALC.EXE. To get the text, I first used GFlags to turn on Show Loader Snaps for CALC.EXE. Next, I ran CALC.EXE under the control of MSDEV.EXE and captured the output from the debug pane.

Loading EXE and DLL Programs

Matt Pietrek

In Figure 6, note that all the output originating in NTDLL is preceded by an LDR: prefix. Other lines in the output (for instance, "Loaded symbols for XXX") were inserted by the MSDEV process. In looking at the LDR: output, there's a wealth of information. For example, as the process starts, the complete path to the EXE is given, along with the current directory and search path.

As NTDLL loads each DLL and fixes up imported functions, you'll see lines like this:

```
LDR: ntdll.dll used by SHELL32.dll
LDR: Snapping imports for SHELL32.dll from ntdll.dll
```

The first line decrees that SHELL32.DLL links to APIs in NTDLL. The second line shows that the imported NTDLL APIs are being "snapped." When an executable module imports functions from another DLL, an array of function pointers resides in the importing module. This array of function pointers is known as the IAT. One of the loader's jobs is to locate the addresses of the imported functions and punch them into the IAT. Hence, the term "snapping" in the LDR: output.

Another interesting set of lines in the output shows bound DLLs being handled:

```
LDR: COMCTL32.dll bound to KERNEL32.dll
LDR: COMCTL32.dll has correct binding to KERNEL32.dll
```

In previous columns, I've talked about the binding process done by BIND.EXE or the BindImageEx API in IMAGEHLP.DLL. Binding an executable to a DLL is the act of looking up the address of the imported APIs and writing them to the importing executable. This speeds up the loading process since the imported addresses don't have to be looked up at load time.

The first line in the above output indicates that the COMCTL32 has bound against APIs in KERNEL32.DLL. The second line indicates that the bound addresses are correct. The loader does this by comparing timestamps. If the timestamps don't match, the binding is invalid. In this case, the loader has to look up the imported addresses just as if the executable hadn't been bound in the first place.

TLS Initialization

I'll finish up this column by showing pseudocode for one other routine. In LdrpRunInitializeRoutines, right before the module's entry point is called, NTDLL checks to see if the module needs TLS initialization. If so, it calls LdrpCallTlsInitializers. Figure 7 shows my pseudocode for this routine.

The code in LdrpCallTlsInitializers is simple enough. Located in the PE header is an offset (RVA) to an IMAGE_TLS_DIRECTORY structure (defined in WINNT.H). The code calls RtlImageDirectoryEntryToData to obtain a pointer to this structure. Within the IMAGE_TLS_DIRECTORY structure is a pointer to an array of callback function addresses to be invoked. These functions are prototyped as IMAGE_TLS_CALLBACK functions, which are defined in WINNT.H. Not coincidentally, the TLS initialization callbacks happen to look just like a DllMain function. For what it's worth, when using __declspec(thread) variables, Visual C++ emits data that causes this routine to be invoked. However, no actual callbacks are currently defined by the runtime library, so the array of function pointers is a single NULL entry.

Conclusion

This wraps up my coverage of Windows NT module initialization. Obviously, I have skipped or skimmed over a lot of related material. For example, what is the algorithm for determining the order in which the modules will be initialized? The algorithm Windows NT uses has changed at least once,

Loading EXE and DLL Programs

Matt Pietrek

and it would be nice to have a Microsoft technical note that at least gives some guidelines. Likewise, I haven't covered the mirror image topic: module unloading. However, I hope this glimpse into the inner workings of the Windows NT loader has provided you with material for further exploration.

Figure 1 RunInit.cpp

```
//=====
// Matt Pietrek, September 1999 Microsoft Systems Journal
// Pseudocode for LdrpRunInitializeRoutines in NTDLL.DLL (Windows NT 4.0, SP3)
// bImplicitLoad parameter is nonzero when LdrpRunInitializeRoutines is
// called for the first time in a process (that is, when the implicitly
// linked modules are initialized.
// On subsequent invocations (caused by calls to LoadLibrary), bImplicitLoad
// is 0.
//=====

#include <ntexapi.h>    // For HardError defines near the end

// Global symbols (name is accurate, and comes from NTDLL.DBG)
// _NtdllBaseTag
// _ShowSnaps
// _SaveSp
// _CurSp
// _LdrpInLdrInit
// _LdrpFatalHardErrorCount
// _LdrpImageHasTls

NTSTATUS
LdrpRunInitializeRoutines( DWORD bImplicitLoad )
{
    // Get the number of modules that *might* need to be initialized.  Some
    // of the modules may already have been initialized.

    unsigned nRoutinesToRun = _LdrpClearLoadInProgress();

    if ( nRoutinesToRun )
    {
        // If there are any init routines, allocate memory for an array
        // containing information about each module
        pInitNodeArray = _RtlAllocateHeap(GetProcessHeap(),
                                         _NtdllBaseTag + 0x60000,
                                         nRoutinesToRun * 4 );

        if ( 0 == pInitNodeArray )    // Make sure allocation worked
            return STATUS_NO_MEMORY;
    }
    else
        pInitNodeArray = 0;

    // The Process Environment Block (Peb), keeps a pointer to the linked
    // list of modules that have just been loaded.  Get a pointer to this info
    //
    pCurrNode = *(pCurrentPeb->ModuleLoaderInfoHead);
    ModuleLoaderInfoHead = pCurrentPeb->ModuleLoaderInfoHead;

    if ( _ShowSnaps )
    {
        _DbgPrint( "LDR: Real INIT LIST\n" );
    }

    nModulesInitatedSoFar = 0;

    if ( pCurrNode != ModuleLoaderInfoHead )
    {
```

Loading EXE and DLL Programs

Matt Pietrek

```
// Iterate through the linked list
//
while ( pCurrNode != ModuleLoaderInfoHead )
{
    ModuleLoaderInfo  pModuleLoaderInfo;

    // Apparently the next node pointer is 0x10 bytes inside
    // the ModuleLoaderInfo structure
    //
    pModuleLoaderInfo = &NextNode - 0x10;

    // This doesn't seem to do anything...
    localVar3C = pModuleLoaderInfo;

    // Determine if the module has already been initialized.  If so,
    // skip over it.
    //
    // X_LOADER_SAW_MODULE = 0x40
    if ( !(pModuleLoaderInfo->Flags35 & X_LOADER_SAW_MODULE) )
    {
        // This module hasn't previously been initialized.  Check
        // to see if it has an EntryPoint to call.
        //
        if ( pModuleLoaderInfo->EntryPoint )
        {
            // This previously uninitialized module has an entry
            // point.  Add it to the array of modules that will
            // be called to initialize later in this routine.
            //
            pInitNodeArray[nModulesInitedSoFar] =pModuleLoaderInfo;

            // If ShowSnaps is nonzero, spit out the module path
            // and the entry point address for the module.  For example:
            // C:\WINNT\system32\KERNEL32.dll init routine 77f01000
            if ( _ShowSnaps )
            {
                _DbgPrint( "%wZ init routine %x\n",
                    &pModuleLoaderInfo->24,
                    pModuleLoaderInfo->EntryPoint );
            }

            nModulesInitedSoFar++;
        }
    }

    // Set the X_LOADER_SAW_MODULE flag for this module.  Note that
    // the module hasn't actually been initialized.  Rather, it will
    // be by the time this routine returns.
    pModuleLoaderInfo->Flags35 &= X_LOADER_SAW_MODULE;

    // Advance to the next node in the module list
    pCurrNode = pCurrNode->pNext
}
}
else
{
    pModuleLoaderInfo = localVar3C;    // May not be initialized???
}

if ( 0 == pInitNodeArray )
    return STATUS_SUCCESS;

// At this point, pInitNodeArray contains an array of pointers to
// modules that haven't yet seen the DLL_PROCESS_ATTACH notification.
// It's now time to start calling the initialization routines.
```

Loading EXE and DLL Programs

Matt Pietrek

```
//
try    // Wrap all this in a try block, in case the init routine faults
{
    nModulesInitedSoFar = 0; // Start at array element 0

    // Begin iterating through the module array
    //
    while ( nModulesInitedSoFar < nRoutinesToRun )
    {
        // Get a pointer to the module's info out of the array
        pModuleLoaderInfo = pInitNodeArray[ nModulesInitedSoFar ];

        // This doesn't seem to do anything...
        localVar3C = pModuleLoaderInfo;

        nModulesInitedSoFar++;

        // Store init routine address in a local variable
        pfnInitRoutine = pModuleLoaderInfo->EntryPoint;

        fBreakOnDllLoad = 0; // Default is to not break on load

        // If this process is a debuggee, check to see if the loader
        // should break into a debugger before calling the initialization.
        //
        // DebuggerPresent (offset 2 in PEB) is what IsDebuggerPresent()
        // returns. IsDebuggerPresent is a Windows NT-only API.
        if ( pCurrentPeb->DebuggerPresent || pCurrentPeb->1 )
        {
            LONG retCode;

            // Query the "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\
            // Windows NT\CurrentVersion\Image File Execution Options"
            // registry key. If a subkey entry with the name of
            // the executable exists, check for the BreakOnDllLoad value.
            retCode = _LdrQueryImageFileExecutionOptions(
                pModuleLoaderInfo->pwszDllName, "BreakOnDllLoad",
                REG_DWORD, &fBreakOnDllLoad,
                sizeof(DWORD), 0 );

            // If reg value not found (usually the case), then don't
            // break on this DLL init
            if ( retCode <= STATUS_SUCCESS )
                fBreakOnDllLoad = 0;
        }

        if ( fBreakOnDllLoad )
        {
            if ( _ShowSnaps )
            {
                // Inform the debug output stream of the module name
                // and the init routine address before actually breaking
                // into the debugger

                _DbgPrint( "LDR: %wZ loaded.",
                    &pModuleLoaderInfo->pModuleLoaderInfo );

                _DbgPrint( "- About to call init routine at %lx\n",
                    pfnInitRoutine )
            }

            // Break into the debugger
            _DbgBreakPoint(); // An INT 3, followed by a RET
        }
        else if ( _ShowSnaps && pfnInitRoutine )
    }
}
```

Loading EXE and DLL Programs

Matt Pietrek

```
{
    // Inform the debug output stream of the module name
    // and the init routine address before calling it
    _DbgPrint( "LDR: %wZ loaded.",
              pModuleLoaderInfo->pModuleLoaderInfo );

    _DbgPrint("- Calling init routine at %lx\n", pfnInitRoutine);
}

if ( pfnInitRoutine )
{
    // Set flag indicating that the DLL_PROCESS_ATTACH notification
    // has been sent to this DLL
    // (Shouldn't this come *after* the actual call?)
    // X_LOADER_CALLED_PROCESS_ATTACH = 0x8
    pModuleLoaderInfo->Flags36 |= X_LOADER_CALLED_PROCESS_ATTACH;

    // If there's Thread Local Storage (TLS) for this module,
    // call the TLS init functions.  *** NOTE *** This only
    // occurs during the first time this code is called (when
    // implicitly loaded DLLs are initialized).  Dynamically
    // loaded DLLs shouldn't use TLS declared vars, as per the
    // SDK documentation
    if ( pModuleLoaderInfo->bHasTLS && bImplicitLoad )
    {
        _LdrpCallTlsInitializers( pModuleLoaderInfo->hModDLL,
                                  DLL_PROCESS_ATTACH );
    }

    hModDLL = pModuleLoaderInfo->hModDLL

    MOV     ESI,ESP // Save off the ESP register into ESI

    MOV     EDI,DWORD PTR [pfnInitRoutine]
           // Load EDI with module's entry point

    // In C++ code, the following ASM would look like:
    // initRetValue =
    // pfnInitRoutine(hInstDLL,DLL_PROCESS_ATTACH,bImplicitLoad);

    PUSH   DWORD PTR [bImplicitLoad]

    PUSH   DLL_PROCESS_ATTACH

    PUSH   DWORD PTR [hModDLL]

    CALL   EDI      // Call the init routine.  Excellent point
                   // to set a breakpoint.  Stepping into this
                   // call will take you to the DLL's entry point

    MOV     BYTE PTR [initRetValue],AL // Save the return value
           // from the entry point

    MOV     DWORD PTR [_SaveSp],ESI // Save stack values after the
    MOV     DWORD PTR [_CurSp],ESP // entry point code returns

    MOV     ESP,ESI // Restore ESP to value before the call

    // Check ESP (stack pointer) after the call.  If it's not
    // the same as the ESP value before the call, the DLL's
    // init routine didn't clean up the stack properly.  For
    // example, it's entry routine may have been defined
    // improperly.  Although this rarely happens, if it does,
    // let the user know and ask if they want to continue.
    if ( _CurSP != _SavSP )
```

Loading EXE and DLL Programs

Matt Pietrek

```
{
    hardErrorParam = pModuleLoaderInfo->FullDllPath;

    hardErrorRetCode = _NtRaiseHardError(
        STATUS_BAD_DLL_ENTRYPOINT | 0x10000000,
        1, // Number of parameters
        1, // UnicodeStringParametersMask,
        &hardErrorParam,
        OptionYesNo, // Let user decide
        &hardErrorResponse );

    if ( _LdrpInLdrInit )
        _LdrpFatalHardErrorCount++;

    if ( (hardErrorRetCode >= STATUS_SUCCESS)
        && (ResponseYes == hardErrorResponse) )
    {
        return STATUS_DLL_INIT_FAILED;
    }
}

// If the DLL's entry point returned 0 (failure), tell the user
if ( 0 == initRetVal )
{
    DWORD hardErrorParam2;
    DWORD hardErrorResponse2;

    hardErrorParam2 = pModuleLoaderInfo->FullDllPath;

    _NtRaiseHardError( STATUS_DLL_INIT_FAILED,
        1, // Number of parameters
        1, // UnicodeStringParametersMask
        &hardErrorParam2,
        OptionOk, // OK is only response
        &hardErrorResponse2 );

    if ( _LdrpInLdrInit )
        _LdrpFatalHardErrorCount++;

    return STATUS_DLL_INIT_FAILED;
}
}

// If the EXE itself has TLS declared vars, call the init routines.
// See the comment for the previous call to _LdrpCallTlsInitializers
// for more details.
//
if ( _LdrpImageHasTls && bImplicitLoad )
{
    _LdrpCallTlsInitializers( pCurrentPeb->ProcessImageBase,
        DLL_PROCESS_ATTACH );
}
}
__finally
{
    // Before exiting the routine, make sure that the memory allocated
    // at the beginning is freed
    _RtlFreeHeap( GetProcessHeap(), 0, pInitNodeArray );
}

return STATUS_SUCCESS;
}
```

Loading EXE and DLL Programs

Matt Pietrek

Figure 6 ShowSnaps Output from CALC.EXE

```
## Matt's comments denoted by ##'s
Loaded 'C:\WINNT\system32\CALC.EXE', no matching symbolic information found.
Loaded symbols for 'C:\WINNT\system32\ntdll.dll'
LDR: PID: 0x3a started - '"C:\WINNT\system32\CALC.EXE"'
LDR: NEW PROCESS
    Image Path: C:\WINNT\system32\CALC.EXE (CALC.EXE)
    Current Directory: C:\WINNT\system32
    Search Path: C:\WINNT\system32;.;C:\WINNT\System32;C:\WINNT\system;...
LDR: SHELL32.dll used by CALC.EXE
Loaded 'C:\WINNT\system32\SHELL32.DLL', no matching symbolic information found.
LDR: ntdll.dll used by SHELL32.dll
LDR: Snapping imports for SHELL32.dll from ntdll.dll
LDR: KERNEL32.dll used by SHELL32.dll
Loaded symbols for 'C:\WINNT\system32\KERNEL32.DLL'
LDR: ntdll.dll used by KERNEL32.dll
LDR: Snapping imports for KERNEL32.dll from ntdll.dll
LDR: Snapping imports for SHELL32.dll from KERNEL32.dll
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlEnterCriticalSection
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlDeleteCriticalSection

//.... additional output omitted

LDR: GDI32.dll used by SHELL32.dll
Loaded symbols for 'C:\WINNT\system32\GDI32.DLL'
LDR: ntdll.dll used by GDI32.dll
LDR: Snapping imports for GDI32.dll from ntdll.dll
LDR: KERNEL32.dll used by GDI32.dll
LDR: Snapping imports for GDI32.dll from KERNEL32.dll
LDR: USER32.dll used by GDI32.dll
Loaded symbols for 'C:\WINNT\system32\USER32.DLL'
LDR: ntdll.dll used by USER32.dll
LDR: Snapping imports for USER32.dll from ntdll.dll
LDR: KERNEL32.dll used by USER32.dll
LDR: Snapping imports for USER32.dll from KERNEL32.dll
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlSizeHeap
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlReAllocateHeap
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlFreeHeap
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlAllocateHeap

//.... additional output omitted

## Note loader looking for and verifying "bound" DLL imports in COMCTL32
Loaded 'C:\WINNT\system32\COMCTL32.DLL', no matching symbolic information found.
LDR: COMCTL32.dll bound to ntdll.dll
LDR: COMCTL32.dll has correct binding to ntdll.dll
LDR: COMCTL32.dll bound to GDI32.dll
LDR: COMCTL32.dll has correct binding to GDI32.dll
LDR: COMCTL32.dll bound to KERNEL32.dll
LDR: COMCTL32.dll has correct binding to KERNEL32.dll
LDR: COMCTL32.dll bound to ntdll.dll via forwarder(s) from KERNEL32.dll
LDR: COMCTL32.dll has correct binding to ntdll.dll
LDR: COMCTL32.dll bound to USER32.dll
LDR: COMCTL32.dll has correct binding to USER32.dll
LDR: COMCTL32.dll bound to ADVAPI32.dll
```

Loading EXE and DLL Programs

Matt Pietrek

LDR: COMCTL32.dll has correct binding to ADVAPI32.dll

//.... additional output omitted

```
LDR: Refcount    COMCTL32.dll (1)
LDR: Refcount    GDI32.dll (3)
LDR: Refcount    KERNEL32.dll (6)
LDR: Refcount    USER32.dll (4)
LDR: Refcount    ADVAPI32.dll (5)
LDR: Refcount    KERNEL32.dll (7)
LDR: Refcount    GDI32.dll (4)
LDR: Refcount    USER32.dll (5)
```

List of implicit link DLLs to be init'ed.

```
LDR: Real INIT LIST
      C:\WINNT\system32\KERNEL32.dll init routine 77f01000
      C:\WINNT\system32\RPCRT4.dll init routine 77e1b6d5
      C:\WINNT\system32\ADVAPI32.dll init routine 77dc1000
      C:\WINNT\system32\USER32.dll init routine 77e78037
      C:\WINNT\system32\COMCTL32.dll init routine 71031a18
      C:\WINNT\system32\SHELL32.dll init routine 77c41094
```

Beginning of actual calls to implicitly linked DLL init routines

```
LDR: KERNEL32.dll loaded. - Calling init routine at 77f01000
LDR: RPCRT4.dll loaded. - Calling init routine at 77e1b6d5
LDR: ADVAPI32.dll loaded. - Calling init routine at 77dc1000
LDR: USER32.dll loaded. - Calling init routine at 77e78037
```

USER32 does AppInit_DLLs thing, so static inits temporarily interrupted

In this case, "globaldll.dll" is LoadLibrary'ed from USER32 init code

```
LDR: LdrLoadDll, loading c:\temp\globaldll.dll from C:\WINNT\system32;.
LDR: Loading (DYNAMIC) c:\temp\globaldll.dll
Loaded 'C:\TEMP\GlobalDLL.dll', no matching symbolic information found.
LDR: KERNEL32.dll used by globaldll.dll
```

//.... additional output omitted

```
LDR: Real INIT LIST
      c:\temp\globaldll.dll init routine 10001310
LDR: globaldll.dll loaded. - Calling init routine at 10001310
```

Now back to calling the inits of the statically linked DLLs

```
LDR: COMCTL32.dll loaded. - Calling init routine at 71031a18
LDR: LdrGetDllHandle, searching for USER32.dll from
LDR: LdrGetProcedureAddress by NAME - GetSystemMetrics
LDR: LdrGetProcedureAddress by NAME - MonitorFromWindow
LDR: SHELL32.dll loaded. - Calling init routine at 77c41094
```

//.... additional output omitted

Figure 7 TLSInit.cpp

```
void _LdrpCallTlsInitializers( HMODULE hModule, DWORD fdwReason )
{
    PIMAGE_TLS_DIRECTORY pTlsDir;
    DWORD size

    // Look up the TLS directory in the IMAGE_OPTIONAL_HEADER.DataDirectory
    pTlsDir = _RtlImageDirectoryEntryToData(hModule,
                                           1,
                                           IMAGE_DIRECTORY_ENTRY_TLS,
                                           &size );

    __try // Protect all this code with a try/catch block
```

Loading EXE and DLL Programs

Matt Pietrek

```
{
if ( pTlsDir->AddressOfCallbacks )
{
    if ( _ShowSnaps )    // diagnostic output
    {
        _DbgPrint( "LDR: Tls Callbacks Found. "
                  "Imagebase %lx Tls %lx CallBacks %lx\n",
                  hModule, TlsDir, pTlsDir->AddressOfCallbacks );
    }

    // Get pointer to beginning of array of TLS callback addresses
    PVOID * pCallbacks = pTlsDir->AddressOfCallbacks;

    while ( *pCallbacks )    // Iterate through each array entry
    {
        PIMAGE_TLS_CALLBACK pTlsCallback = *pCallbacks;
        pCallbacks++;

        if ( _ShowSnaps )    // More diagnostic output
        {
            _DbgPrint( "LDR: Calling Tls Callback "
                      "Imagebase %lx Function %lx\n",
                      hModule, pTlsCallback );
        }

        // Make the actual call
        pTlsCallback( hModule, fdwReason, 0 );
    }
}
__except( EXCEPTION_EXECUTE_HANDLER )
{
}
}
```