

## Kernel User-Mode Debugging Support (Dbgk)

The final piece of the puzzle lives in kernel-mode, and provides the events and structures that we have seen until now so that debugging can work at all. Dbgk does not rely on KD, and is an entirely different component which, since Windows XP, provides its own object and system calls to manage it. Previous versions of Windows did not have such an object, and relied instead on a static data structure that would be analyzed in memory and then used for various notifications sent through Windows's Local Procedure Call (LPC) mechanisms.

One of the great things about the availability of these system calls and the presence of a debug object is the fact that kernel-mode drivers can also engage in user-mode debugging. Although this was probably not one of the goals of this new design, it is a feature that should be of interest to some. Although the actual Nt\* calls are not exported, they can still be accessed by a driver which knows their system call ID. Even though this number changes between each OS version, it is relatively easy to keep a table in the driver. By adding a TDI interface to such a driver, it could be possible to develop a high-speed remote debugger driver, which would have no user-mode components at all and thus allow debugging every single process on the machine remotely.

The first thing we'll do is take a look at the actual object which implements user-mode debugging, the `DEBUG_OBJECT`:

```
//  
// Debug Object  
//  
typedef struct _DEBUG_OBJECT  
{  
    KEVENT EventsPresent;  
    FAST_MUTEX Mutex;  
    LIST_ENTRY EventList;  
    union  
    {  
        ULONG Flags;  
        struct  
        {  
            UCHAR DebuggerInactive:1;  
            UCHAR KillProcessOnExit:1;  
        };  
    };  
} DEBUG_OBJECT, *PDEBUG_OBJECT;
```

As you can see, the object itself is a rather lightweight wrapper around the actual event on which user-mode uses `WaitForDebugEvent`, a list of actual debug events, a lock, and

certain flags relating to this debugging session, such as whether or not the debugger is connected, and if the process should be killed when disconnecting. Therefore, the structure we are more interested in is the `DEBUG_EVENT` structure:

```
//  
// Debug Event  
//  
typedef struct _DEBUG_EVENT  
{  
    LIST_ENTRY EventList;  
    KEVENT ContinueEvent;  
    CLIENT_ID ClientId;  
    PEPROCESS Process;  
    PETHREAD Thread;  
    NTSTATUS Status;  
    ULONG Flags;  
    PETHREAD BackoutThread;  
    DBGKM_MSG ApiMsg;  
} DEBUG_EVENT, *PDEBUG_EVENT;
```

It's this structure that contains all the data related to a debugging event. Since many events can be queued up before a caller does a `WaitForDebugEvent` in user-mode, debug events must be linked together with the `DEBUG_OBJECT`, and that's what the event list is for.

Some of the other members hold the PID and TID of the event from which the notification came, as well as pointers to the respective process and thread objects in kernel-mode. The event in this structure is used internally to notify the kernel when a response to a debugger message is available. This response usually comes in the form of a `ContinueDebugEvent` call from Win32, which will signal the event.

The final structure contained in a debug event is the actual API message being sent, which contains the data that user-mode will see, and is the kernel-mode representation of the `DBGUI_WAIT_STATE_CHANGE` structure. That's right, the kernel has yet *another* way of representing debug events, and it too will need to be converted later so that the Native Debugging interface can understand it.

The good thing, as seen in the structure below, is that most of the fields have remained constant, and that the kernel internally still uses the `DBGKM` structure which were already shown in the `DbgUi` structure. However, instead of using `DBG_STATE` constants, the kernel uses another type of constants called API Message Numbers, which are shown below:

```
//  
// Debug Message API Number  
//
```

```

typedef enum _DBGKM_APINUMBER
{
    DbgKmExceptionApi = 0,
    DbgKmCreateThreadApi = 1,
    DbgKmCreateProcessApi = 2,
    DbgKmExitThreadApi = 3,
    DbgKmExitProcessApi = 4,
    DbgKmLoadDllApi = 5,
    DbgKmUnloadDllApi = 6,
    DbgKmErrorReportApi = 7,
    DbgKmMaxApiNumber = 8,
} DBGKM_APINUMBER;

```

These API Numbers are self-explanatory and are still kept for compatibility with the old LPC mechanism. The kernel will convert them to the actual debug states expected by DbgUi. Now let's look at the Debug Message structure itself, which matches the same message used in previous versions of windows on top of the LPC mechanism:

```

//
// LPC Debug Message
//
typedef struct _DBGKM_MSG
{
    PORT_MESSAGE h;
    DBGKM_APINUMBER ApiNumber;
    ULONG ReturnedStatus;
    union
    {
        DBGKM_EXCEPTION Exception;
        DBGKM_CREATE_THREAD CreateThread;
        DBGKM_CREATE_PROCESS CreateProcess;
        DBGKM_EXIT_THREAD ExitThread;
        DBGKM_EXIT_PROCESS ExitProcess;
        DBGKM_LOAD_DLL LoadDll;
        DBGKM_UNLOAD_DLL UnloadDll;
    };
} DBGKM_MSG, *PDBGKM_MSG;

```

Note of course, that for our purposes, we can ignore the PORT\_MESSAGE part of the structure, as we will not be looking at DbgSs (the component that handles LPC Messages and the layer that wrapped DbgUi before Windows XP).

Now that we know what the structures look at, we can start looking at some of the Native API functions (System calls) which wrap the Debug Object as if it was just another object such as an event or semaphore.

The first system call that is required and that we've seen from DbgUi in Part 2 is NtCreateDebugObject, which will return a handle to the debug object that can be attached and waited on later. The implementation is rather simple:

```

NTSTATUS
NTAPI
NtCreateDebugObject(OUT PHANDLE DebugHandle,
                   IN ACCESS_MASK DesiredAccess,
                   IN POBJECT_ATTRIBUTES ObjectAttributes,
                   IN BOOLEAN KillProcessOnExit)
{
    KPROCESSOR_MODE PreviousMode = ExGetPreviousMode();
    PDEBUG_OBJECT DebugObject;
    HANDLE hDebug;
    NTSTATUS Status = STATUS_SUCCESS;
    PAGED_CODE();

    /* Check if we were called from user mode */
    if (PreviousMode != KernelMode)
    {
        /* Enter SEH for probing */
        _SEH_TRY
        {
            /* Probe the handle */
            ProbeForWriteHandle(DebugHandle);
        }
        _SEH_HANDLE
        {
            /* Get exception error */
            Status = _SEH_GetExceptionCode();
        } _SEH_END;
        if (!NT_SUCCESS(Status)) return Status;
    }

    /* Create the Object */
    Status = ObCreateObject(PreviousMode,
                           DbgkDebugObjectType,
                           ObjectAttributes,
                           PreviousMode,
                           NULL,
                           sizeof(DEBUG_OBJECT),
                           0,
                           0,
                           (PVOID*)&DebugObject);
    if (NT_SUCCESS(Status))
    {
        /* Initialize the Debug Object's Fast Mutex */
        ExInitializeFastMutex(&DebugObject->Mutex);

        /* Initialize the State Event List */
        InitializeListHead(&DebugObject->EventList);

        /* Initialize the Debug Object's Wait Event */
        KeInitializeEvent(&DebugObject->EventsPresent,
                        NotificationEvent,
                        FALSE);

        /* Set the Flags */
        DebugObject->KillProcessOnExit = KillProcessOnExit;
    }
}

```

```

/* Insert it */
Status = ObInsertObject((PVOID)DebugObject,
                        NULL,
                        DesiredAccess,
                        0,
                        NULL,
                        &hDebug);

if (NT_SUCCESS(Status))
{
    _SEH_TRY
    {
        *DebugHandle = hDebug;
    }
    _SEH_HANDLE
    {
        Status = _SEH_GetExceptionCode();
    } _SEH_END;
}

/* Return Status */
DBGKTRACE(DBGK_OBJECT_DEBUG, "Handle: %p DebugObject: %p\n",
          hDebug, DebugObject);
return Status;

```

One of the interesting things in using this API directly from user-mode is that it allows naming the debug object so that it can be inserted in the object directory namespace. Unfortunately, there does not exist an `NtOpenDebugObject` call, so the name cannot be used for lookups, but this can be stored internally, or the objects can be inserted into a tree such as `\DebugObjects` which can later be enumerated.

Mixed with the fact that the `DbgUi` layer can be thus skipped, this means that debug object handles need not be stored in the TEB, and this gives the debugger writer the ability to write a debugger which is debugging multiple processes in the same time, seamlessly switching between debug objects, and implementing a custom `WaitForDebugEvent` by using `WaitForMultipleObjects` which can receive debugging events from multiple processes.

Handling the messages and handles from all these processes can be slightly daunting, but using a model similar to how `kernel32` stores per-thread data in the TEB, it can be modeled to additionally store per-process data. The end result would be a powerful debugger an innovation that others don't yet support.

Now let's look at actual attachment to a process, which is done by `NtDebugActiveProcess`:

```

NTSTATUS
NTAPI
NtDebugActiveProcess(IN HANDLE ProcessHandle,
                    IN HANDLE DebugHandle)
{
    PEPROCESS Process;
    PDEBUG_OBJECT DebugObject;
    KPROCESSOR_MODE PreviousMode = KeGetPreviousMode();
    PETHREAD LastThread;
    NTSTATUS Status;
    PAGED_CODE();
    DBGKTRACE(DBGK_PROCESS_DEBUG, "Process: %p Handle: %p\n",
              ProcessHandle, DebugHandle);

    /* Reference the process */
    Status = ObReferenceObjectByHandle(ProcessHandle,
                                       PROCESS_SUSPEND_RESUME,
                                       PsProcessType,
                                       PreviousMode,
                                       (PVOID*)&Process,
                                       NULL);
    if (!NT_SUCCESS(Status)) return Status;

    /* Don't allow debugging the initial system process */
    if (Process == PsInitialSystemProcess) return STATUS_ACCESS_DENIED;

    /* Reference the debug object */
    Status = ObReferenceObjectByHandle(DebugHandle,
                                       DEBUG_OBJECT_ADD_REMOVE_PROCESS,
                                       DbgkDebugObjectType,
                                       PreviousMode,
                                       (PVOID*)&DebugObject,
                                       NULL);

    if (!NT_SUCCESS(Status))
    {
        /* Dereference the process and exit */
        ObDereferenceObject(Process);
        return Status;
    }

    /* Acquire process rundown protection */
    if (!ExAcquireRundownProtection(&Process->RundownProtect))
    {
        /* Dereference the process and debug object and exit */
        ObDereferenceObject(Process);
        ObDereferenceObject(DebugObject);
        return STATUS_PROCESS_IS_TERMINATING;
    }

    /* Send fake create messages for debuggers to have a consistent state */
    Status = DbgkPostFakeProcessCreateMessages(Process,
                                               DebugObject,
                                               &LastThread);
    Status = DbgkSetProcessDebugObject(Process,
                                       DebugObject,

```

```

        Status,
        LastThread);

    /* Release rundown protection */
    ExReleaseRundownProtection(&Process->RundownProtect);

    /* Dereference the process and debug object and return status */
    ObDereferenceObject(Process);
    ObDereferenceObject(DebugObject);
    return Status;
}

```

This API is also simple, and relies on much larger internal routines to perform most of the work. First of all, one of the problems with attaching is that the process might have already created multiple new threads, as well as loaded various DLLs. The system cannot anticipate which process will be debugged, so it does not queue these debug events anywhere, instead, the Dbgk module must scan each each thread and module, and send the appropriate “fake” event message to the debugger. For example, when attaching to a process, this is what will generate the storm of DLL load events so that the debugger can know what’s going on.

Without going in the internals of the Dbgkp calls, the way that DLL load messages are sent are by looping the loader list contained in the PEB, through Peb->Ldr. There is a hard-coded maximum of 500 DLLs, so that the list won’t be looped indefinitely. Instead of using the DLL name contained in the corresponding LDR\_DATA\_TABLE\_ENTRY structures however, an internal API called MmGetFileNameForAddress is used, which will find the VAD for the DLL’s base address, and use it to get the SECTION\_OBJECT associated with it. From this SECTION\_OBJECT, the Memory Manager can find the FILE\_OBJECT, and then use ObQueryNameString to query the full name of the DLL, which can be used to open the handle that user-mode will receive.

Note that the NamePointer parameter of the Load DLL structure is not filled out, although it easily could be.

For looping newly created threads, the helper PsGetNextProcessThread API is used, which will loop every thread. For the first thread, this will generate a Create Process debug event, while each subsequent thread will result in Create Thread messages. For the process, event data is retrieved from the SectionBaseAddress pointer, which has the base image pointer. For threads, the only data returned is the start address, saved already in ETHREAD.

Finally, the DbgkpSetProcessDebugObject does the complicated work of associating the object with the process. Firstly, there exists a possibility that even newer threads were

created after the initial list was parsed in DbgPostFakeThreadMessages. This routine will therefore actually acquire the Debug Port Mutex and call DbgPostFakeThreadMessages again, to catch any threads that were missed. Logically, this could result in the same message being sent twice, but one of the ETHREAD flags comes into play: CreateThreadReported. DbgPostFakeThreadMessages will check for this flag before sending a message, so duplicates are avoided (the same applies for EPROCESS).

The second part of DbgkSetProcessDebugObject will parse any debug events that have already been associated to the object. This means all those fake messages we just sent. This will mean acquiring the rundown protection for each thread, as well as checking for various race conditions or not-fully-inserted threads which might've been picked up. Finally, the PEB is modified so that the BeingDebugged flag is enabled. Once the routine is done, the debug object is fully associated to the target.

Now let's look at the analogous routine is implemented, NtRemoveProcessDebug, which allows to detach from an actively debugged process.

```
NTSTATUS
NTAPI
NtRemoveProcessDebug(IN HANDLE ProcessHandle,
                    IN HANDLE DebugHandle)
{
    PEPROCESS Process;
    PDEBUG_OBJECT DebugObject;
    KPROCESSOR_MODE PreviousMode = KeGetPreviousMode();
    NTSTATUS Status;
    PAGED_CODE();
    DBGKTRACE(DBGK_PROCESS_DEBUG, "Process: %p Handle: %p\n",
              ProcessHandle, DebugHandle);

    /* Reference the process */
    Status = ObReferenceObjectByHandle(ProcessHandle,
                                       PROCESS_SUSPEND_RESUME,
                                       PsProcessType,
                                       PreviousMode,
                                       (PVOID*)&Process,
                                       NULL);
    if (!NT_SUCCESS(Status)) return Status;

    /* Reference the debug object */
    Status = ObReferenceObjectByHandle(DebugHandle,
                                       DEBUG_OBJECT_ADD_REMOVE_PROCESS,
                                       DbgkDebugObjectType,
                                       PreviousMode,
                                       (PVOID*)&DebugObject,
                                       NULL);

    if (!NT_SUCCESS(Status))
    {
```



```

/* Check if the caller wanted the return length */
if (ReturnLength)
{
    /* Enter SEH for probe */
    _SEH_TRY
    {
        /* Return required length to user-mode */
        ProbeForWriteUlong(ReturnLength);
        *ReturnLength = sizeof(*DebugInfo);
    }
    _SEH_EXCEPT(_SEH_ExSystemExceptionFilter)
    {
        /* Get SEH Exception code */
        Status = _SEH_GetExceptionCode();
    }
    _SEH_END;
}
if (!NT_SUCCESS(Status)) return Status;

/* Open the Object */
Status = ObReferenceObjectByHandle(DebugHandle,
                                   DEBUG_OBJECT_WAIT_STATE_CHANGE,
                                   DbgkDebugObjectType,
                                   PreviousMode,
                                   (PVOID*)&DebugObject,
                                   NULL);

if (NT_SUCCESS(Status))
{
    /* Acquire the object */
    ExAcquireFastMutex(&DebugObject->Mutex);

    /* Set the proper flag */
    if (DebugInfo->KillProcessOnExit)
    {
        /* Enable killing the process */
        DebugObject->KillProcessOnExit = TRUE;
    }
    else
    {
        /* Disable */
        DebugObject->KillProcessOnExit = FALSE;
    }

    /* Release the mutex */
    ExReleaseFastMutex(&DebugObject->Mutex);

    /* Release the Object */
    ObDereferenceObject(DebugObject);
}

/* Return Status */
return Status;
}

```

Finally, the last functionality provided by the debug object is the wait and continue calls, which implement the bi-directional channel through which a debugger can receive events, modify the target state or its own internal data, and then resume execution. These calls are a bit more involved because of the inherent syncing issues involved. First, let's explore NtWaitForDebugEvent:

```

NTSTATUS
NTAPI
NtWaitForDebugEvent(IN HANDLE DebugHandle,
                   IN BOOLEAN Alertable,
                   IN PLARGE_INTEGER Timeout OPTIONAL,
                   OUT PDBGUI_WAIT_STATE_CHANGE StateChange)
{
    KPROCESSOR_MODE PreviousMode = ExGetPreviousMode();
    LARGE_INTEGER SafeTimeout;
    PEPROCESS Process;
    LARGE_INTEGER StartTime;
    PETHREAD Thread;
    BOOLEAN GotEvent;
    LARGE_INTEGER NewTime;
    PDEBUG_OBJECT DebugObject;
    DBGUI_WAIT_STATE_CHANGE WaitStateChange;
    NTSTATUS Status = STATUS_SUCCESS;
    PDEBUG_EVENT DebugEvent, DebugEvent2;
    PLIST_ENTRY ListHead, NextEntry, NextEntry2;
    PAGED_CODE();
    DBGKTRACE(DBGK_OBJECT_DEBUG, "Handle: %p\n", DebugHandle);

    /* Clear the initial wait state change structure */
    RtlZeroMemory(&WaitStateChange, sizeof(WaitStateChange));

    /* Protect probe in SEH */
    _SEH_TRY
    {
        /* Check if we came with a timeout */
        if (Timeout)
        {
            /* Check if the call was from user mode */
            if (PreviousMode != KernelMode)
            {
                /* Probe it */
                ProbeForReadLargeInteger(Timeout);
            }

            /* Make a local copy */
            SafeTimeout = *Timeout;
            Timeout = &SafeTimeout;

            /* Query the current time */
            KeQuerySystemTime(&StartTime);
        }

        /* Check if the call was from user mode */
    }
}

```

```

    if (PreviousMode != KernelMode)
    {
        /* Probe the state change structure */
        ProbeForWrite(StateChange, sizeof(*StateChange), sizeof(ULONG));
    }
}
_SEH_HANDLE
{
    /* Get the exception code */
    Status = _SEH_GetExceptionCode();
}
_SEH_END;
if (!NT_SUCCESS(Status)) return Status;

/* Get the debug object */
Status = ObReferenceObjectByHandle(DebugHandle,
                                   DEBUG_OBJECT_WAIT_STATE_CHANGE,
                                   DbgkDebugObjectType,
                                   PreviousMode,
                                   (PVOID*)&DebugObject,
                                   NULL);
if (!NT_SUCCESS(Status)) return Status;

/* Clear process and thread */
Process = NULL;
Thread = NULL;

/* Start wait loop */
while (TRUE)
{
    /* Wait on the debug object given to us */
    Status = KeWaitForSingleObject(DebugObject,
                                   Executive,
                                   PreviousMode,
                                   Alertable,
                                   Timeout);

    if (!NT_SUCCESS(Status) ||
        (Status == STATUS_TIMEOUT) ||
        (Status == STATUS_ALERTED) ||
        (Status == STATUS_USER_APC))
    {
        /* Break out the wait */
        break;
    }

    /* Lock the object */
    GotEvent = FALSE;
    ExAcquireFastMutex(&DebugObject->Mutex);

    /* Check if a debugger is connected */
    if (DebugObject->DebuggerInactive)
    {
        /* Not connected */
        Status = STATUS_DEBUGGER_INACTIVE;
    }
}

```

```

else
{
    /* Loop the events */
    ListHead = &DebugObject->EventList;
    NextEntry = ListHead->Flink;
    while (ListHead != NextEntry)
    {
        /* Get the debug event */
        DebugEvent = CONTAINING_RECORD(NextEntry,
                                        DEBUG_EVENT,
                                        EventList);
        DBGKTRACE(DBGK_PROCESS_DEBUG, "DebugEvent: %p Flags: %lx\n",
                  DebugEvent, DebugEvent->Flags);

        /* Check flags */
        if (!(DebugEvent->Flags &
              (DEBUG_EVENT_FLAGS_USED | DEBUG_EVENT_FLAGS_INACTIVE)))
        {
            /* We got an event */
            GotEvent = TRUE;

            /* Loop the list internally */
            NextEntry2 = DebugObject->EventList.Flink;
            while (NextEntry2 != NextEntry)
            {
                /* Get the debug event */
                DebugEvent2 = CONTAINING_RECORD(NextEntry2,
                                                DEBUG_EVENT,
                                                EventList);

                /* Try to match process IDs */
                if (DebugEvent2->ClientId.UniqueProcess ==
                    DebugEvent->ClientId.UniqueProcess)
                {
                    /* Found it, break out */
                    DebugEvent->Flags |= DEBUG_EVENT_FLAGS_USED;
                    DebugEvent->BackoutThread = NULL;
                    GotEvent = FALSE;
                    break;
                }

                /* Move to the next entry */
                NextEntry2 = NextEntry2->Flink;
            }

            /* Check if we still have a valid event */
            if (GotEvent) break;
        }

        /* Move to the next entry */
        NextEntry = NextEntry->Flink;
    }

    /* Check if we have an event */
    if (GotEvent)

```

```

{
    /* Save and reference the process and thread */
    Process = DebugEvent->Process;
    Thread = DebugEvent->Thread;
    ObReferenceObject(Process);
    ObReferenceObject(Thread);

    /* Convert to user-mode structure */
    DbgkpConvertKernelToUserStateChange(&WaitStateChange,
                                        DebugEvent);

    /* Set flag */
    DebugEvent->Flags |= DEBUG_EVENT_FLAGS_INACTIVE;
}
else
{
    /* Unsignal the event */
    KeClearEvent(&DebugObject->EventsPresent);
}

/* Set success */
Status = STATUS_SUCCESS;
}

/* Release the mutex */
ExReleaseFastMutex(&DebugObject->Mutex);
if (!NT_SUCCESS(Status)) break;

/* Check if we got an event */
if (!GotEvent)
{
    /* Check if we can wait again */
    if (SafeTimeout.QuadPart < 0)
    {
        /* Query the new time */
        KeQuerySystemTime(&NewTime);

        /* Subtract times */
        SafeTimeout.QuadPart += (NewTime.QuadPart -
StartTime.QuadPart);
        StartTime = NewTime;

        /* Check if we've timed out */
        if (SafeTimeout.QuadPart > 0)
        {
            /* We have, break out of the loop */
            Status = STATUS_TIMEOUT;
            break;
        }
    }
}
else
{
    /* Open the handles and dereference the objects */
    DbgkpOpenHandles(&WaitStateChange, Process, Thread);
}
}

```

```

        ObDereferenceObject(Process);
        ObDereferenceObject(Thread);
        break;
    }
}

/* We're done, dereference the object */
ObDereferenceObject(DebugObject);

/* Protect write with SEH */
_SEH_TRY
{
    /* Return our wait state change structure */
    RtlCopyMemory(StateChange,
                  &WaitStateChange,
                  sizeof(DBGUI_WAIT_STATE_CHANGE));
}
_SEH_EXCEPT(_SEH_ExSystemExceptionFilter)
{
    /* Get SEH Exception code */
    Status = _SEH_GetExceptionCode();
}
_SEH_END;

/* Return status */
return Status;
}

```

The first thing that happens is that a wait on the debug object is done, or more specifically, on the EventsPresent event. When this wait is satisfied, the object is locked and the API first makes sure that it hasn't become inactive before the lock was acquired. Once this is confirmed, the current debug events are parsed, and the system call ensures that the debug event hasn't already been used (processed) and that it hasn't been made inactive. If these flags check out, then the list is parsed again to make sure there aren't any other events for the same process. If any are found, then this event is marked as inactive, and nothing is sent. This seems to block any multiple events from being sent if they're for the same process.

Once a debug event has been obtained, the process and thread are referenced and the structure is converted to a DbgUi Wait State Change structure, and the event is marked as used. After the debug object lock is released, DbgkOpenHandles is called in the success case when a debug event is found, which will open the right handles that user-mode expects in the DbgUi structure, after which the extra references to the process and thread are dropped.

After the wait is done, the DbgUi structure is copied back to the caller.

The final API, NtDebugContinue allows continuing from a debug event, and its implementation serves to remove the debug event sent and to wake the target. It is implemented like this:

```
NTSTATUS
NTAPI
NtDebugContinue(IN HANDLE DebugHandle,
               IN PCLIENT_ID AppClientId,
               IN NTSTATUS ContinueStatus)
{
    KPROCESSOR_MODE PreviousMode = ExGetPreviousMode();
    PDEBUG_OBJECT DebugObject;
    NTSTATUS Status = STATUS_SUCCESS;
    PDEBUG_EVENT DebugEvent = NULL, DebugEventToWake = NULL;
    PLIST_ENTRY ListHead, NextEntry;
    BOOLEAN NeedsWake = FALSE;
    CLIENT_ID ClientId;
    PAGED_CODE();
    DBGKTRACE(DBGK_OBJECT_DEBUG, "Handle: %p Status: %p\n",
             DebugHandle, ContinueStatus);

    /* Check if we were called from user mode */
    if (PreviousMode != KernelMode)
    {
        /* Enter SEH for probing */
        _SEH_TRY
        {
            /* Probe the handle */
            ProbeForRead(AppClientId, sizeof(CLIENT_ID), sizeof(ULONG));
            ClientId = *AppClientId;
            AppClientId = &ClientId;
        }
        _SEH_HANDLE
        {
            /* Get exception error */
            Status = _SEH_GetExceptionCode();
        } _SEH_END;
        if (!NT_SUCCESS(Status)) return Status;
    }

    /* Make sure that the status is valid */
    if ((ContinueStatus != DBG_CONTINUE) &&
        (ContinueStatus != DBG_EXCEPTION_HANDLED) &&
        (ContinueStatus != DBG_EXCEPTION_NOT_HANDLED) &&
        (ContinueStatus != DBG_TERMINATE_THREAD) &&
        (ContinueStatus != DBG_TERMINATE_PROCESS))
    {
        /* Invalid status */
        Status = STATUS_INVALID_PARAMETER;
    }
    else
    {
        /* Get the debug object */
        Status = ObReferenceObjectByHandle(DebugHandle,
```

```

                                DEBUG_OBJECT_WAIT_STATE_CHANGE,
                                DbgkDebugObjectType,
                                PreviousMode,
                                (PVOID*)&DebugObject,
                                NULL);
if (NT_SUCCESS(Status))
{
    /* Acquire the mutex */
    ExAcquireFastMutex(&DebugObject->Mutex);

    /* Loop the state list */
    ListHead = &DebugObject->EventList;
    NextEntry = ListHead->Flink;
    while (ListHead != NextEntry)
    {
        /* Get the current debug event */
        DebugEvent = CONTAINING_RECORD(NextEntry,
                                        DEBUG_EVENT,
                                        EventList);

        /* Compare process ID */
        if (DebugEvent->ClientId.UniqueProcess ==
            AppClientId->UniqueProcess)
        {
            /* Check if we already found a match */
            if (NeedsWake)
            {
                /* Wake it up and break out */
                DebugEvent->Flags &= ~DEBUG_EVENT_FLAGS_USED;
                KeSetEvent(&DebugEvent->ContinueEvent,
                            IO_NO_INCREMENT,
                            FALSE);

                break;
            }

            /* Compare thread ID and flag */
            if ((DebugEvent->ClientId.UniqueThread ==
                AppClientId->UniqueThread) &&
                (DebugEvent->Flags & DEBUG_EVENT_FLAGS_INACTIVE))
            {
                /* Remove the event from the list */
                RemoveEntryList(NextEntry);

                /* Remember who to wake */
                NeedsWake = TRUE;
                DebugEventToWake = DebugEvent;
            }
        }

        /* Go to the next entry */
        NextEntry = NextEntry->Flink;
    }

    /* Release the mutex */
    ExReleaseFastMutex(&DebugObject->Mutex);
}

```

```

    /* Dereference the object */
    ObDereferenceObject(DebugObject);

    /* Check if need a wait */
    if (NeedsWake)
    {
        /* Set the continue status */
        DebugEvent->ApiMsg.ReturnedStatus = Status;
        DebugEvent->Status = STATUS_SUCCESS;

        /* Wake the target */
        DbgkpWakeTarget(DebugEvent);
    }
    else
    {
        /* Fail */
        Status = STATUS_INVALID_PARAMETER;
    }
}

/* Return status */
return Status;
}

```

First, it is responsible for validating the continuation status that is used, to a limited number of recognized status codes. Then, each debug event is looped. If the process and thread IDs match, then the debug event is removed from the list, and the routine remembers that an event was found. If any additional events are found for the same process, then the inactive flag is remove. Recall that this flag was added by the wait routine, to prohibit multiple events for the same process to be sent together.

Once all debug events are parsed, then the target is woken, which basically means resuming the thread if the message was for a fake thread create message, releasing its rundown protection, and then either freeing the debug event or notify whoever was waiting on it (depending on how it was created).

This concludes the last section on the User-Mode Debugging implementation Windows XP and higher, and since this section focused on kernel-mode, let's see what we've learned from this section:

Dbgk is the component in the kernel that handles all the support code for the debugging functionality.

The implementation is exposed through an NT Object called `DEBUG_OBJECT` and provides various system calls to access it.

It is possible to write a kernel-mode debugger for user-mode applications.

It is possible to write a debugger that can debug multiple applications in the same time, as long as the DbgUi layer is skipped and re-implemented using system calls by hand.

The kernel uses its own version of the wait state change structure, encapsulated in a `DEBUG_EVENT` structure.

The kernel still has legacy support for LPC based DbgSs debugging.

The kernel opens all the handles that are present in the event structures, and user-mode is responsible to close them.

The kernel is written such that only one event for the same process is sent at one time.

The kernel needs to parse the PEB Loader Data to get a list of loaded DLLs, and has a hard coded limit of 500 loop iterations.

This also brings us to the end of this series on debugging internals, and the author hopes that readers have enjoyed this thorough analysis and will be able to write better, more flexible, more powerful debuggers which can utilize some of these internals to provide greater power to users and developers.