Portable Systems Group

NT OS/2 Debug Architecture

Author: *Mark Lucovsky*

Revision 1.1, May 8, 1990
Original Draft February 15, 1990

1. Overview 1
1.1 Debug Event Flow 1

2. Debug Architecture Partitioning 2
2.1 Debug Event Generation 3
2.1.1 Event Generation Message Formats 3
2.1.1.1 Exception 4
2.1.1.2 CreateThread 4
2.1.1.3 CreateProcess 5
2.1.1.4 ExitThread 5
2.1.1.5 ExitProcess 5
2.1.1.6 MapSection 6
2.1.1.7 UnMapSection 6
2.2 Event Propagation 6
2.2.1 Emulation Subsystem APIs for Event Propagation 7
2.2.1.1 DbgSsInitialize 7
2.2.1.2 UiLookupRoutine 8
2.2.1.3 SubsystemKeyLookupRoutine 8
2.2.1.4 KmApiMsgFilter 9
2.2.1.5 DbgSsHandleKmApiMsg 10
2.2.2 Event Propagation Message Formats 10
2.2.2.1 Exception 11
2.2.2.2 CreateThread 11
2.2.2.3 CreateProcess 11
2.2.2.4 ExitThread 11
2.2.2.5 ExitProcess 12
2.2.2.6 MapSection 12
2.2.2.7 UnMapSection 12
2.3 Coordinate Debugger and Debuggee 12
2.3.1 Dbg Server Data Structures 12
2.3.1.1 Subsystem Structure 12
2.3.1.2 User Interface Structure 13
2.3.1.3 Application Process Structure 14
2.3.1.4 Application Thread Structure 15
2.3.2 Dbg Server Responses to Debug Event Propagation 16
2.3.2.1 Exception 16
2.3.2.2 CreateThread 16
2.3.2.3 CreateProcess 17
2.3.2.4 ExitThread 17
2.3.2.5 ExitProcess 18
2.3.2.6 MapSection 18
2.3.2.7 UnMapSection 18
2.4 User Interface Interactions with the Dbg Server 18
2.4.1 DbgUiConnectToDbg 19

2.4.2 DbgUiWaitStateChange 19
2.4.2.1 State Change Record 20
2.4.2.1.1 DbgCreateThreadStateChange 20
2.4.2.1.2 DbgCreateProcessStateChange 21
2.4.2.1.3 DbgExitThreadStateChange 22
2.4.2.1.4 DbgExitProcessStateChange 22
2.4.2.1.5 DbgExceptionStateChange 22
2.4.2.1.6 DbgBreakpointStateChange 22
2.4.2.1.7 DbgSingleStepStateChange 22
2.4.2.1.8 DbgMapSectionStateChange 23
2.4.2.1.9 DbgUnMapSectionStateChange 23
2.4.3 DbgUiContinue 23
2.4.3.1 DbgExitThreadStateChange 24
2.4.3.2 DbgExitProcessStateChange 24

1. Overview

This specification describes the Debug Architecture found in NT OS/2. The Debug Architecture consists of the following:

- Dbgk executive component. This component is responsible for generating debug events and sending a message through a processes debug port when a debug event for the process occurs.
- The debugger user interface (DebugUi) is an application that provides the human interface for a debugger. An instance of DebugUi exists for each application being debugged.
- Dbg user-mode subsystem (Dbg server). This component acts as a debug event coordinator, ensuring that debug events occuring in a process are made available to its controlling DebugUi.
- DbgSs APIs. This set of APIs allows an Emulation Subsystem to participate in the NT OS/2 Debug Architecture. While not required, It is expected that Emulation Subsystems pick up debug events coming from the Dbgk component, add information to these events, and then forward the event to the Dbg subsystem.
- DbgUi APIs. This set of APIs allow a DebugUi to communicate with the Dbg server so that it may receive notification of outstanding debug events, and respond to received debug events.

1.1 Debug Event Flow

Before going any further, the following example of a debug event illustrates the typical interaction between the components listed above.

- The OS/2 subsystem (OS2SS) is controlling an application process. The process was started as a debugged process through the **SbCreateForeignSession** API. Upon reciept of the process by OS2SS, a debug port was assigned to the process using **NtSetInformationProcess**. OS2SS owns the processes debug port and receives all messages arriving at this port. The application process contians a single thread. At some point in the thread's lifetime, a reference to inaccessible memory is made. This generates an access violation exception triggering a potential debug event.
- The exception dispatcher in the NT OS/2 executive calls into the Dbgk component (at its **DbgkForwardException** entrypoint) to report the access violation. Dbgk determines whether or not the process has an associated DebugPort. In this case, the process does have a DebugPort, so a DBGKM_EXCEPTION message is formatted. All threads (except for the current thread) in the process are frozen using **KeFreezeThread**. The current thread sends the exception message through its DebugPort and awaits a reply.
- OS2SS receives the exception message. Since the message type stored in the message header is LPC_DEBUG_EVENT, OS2SS calls **DbgSsHandleKmApiMsg** passing the address of the message. **Exception** messages requires no additional information from the OS2SS, so the

message is forwarded to the Dbg server for processing. The message is sent as a datagram so that OS2SS does not have to burn a thread while waiting for a reply.

- The Dbg server receives the exception message. Using the Client Id from the the original exception message, Dbg locates an internal per-thread data structure. The exception message is captured into this data structure. A state change database entry for the application is recorded. The DebugUi for the thread is located, and its debug state change semaphore is signaled (this semaphore is shared between a DebugUi and the Dbg server).

- At this point in time it is important to note that while three threads have participated so far, only one thread remains blocked. The blocked thread is the application thread that caused the access violation. This thread is waiting for a reply to its original exception message.

- At some point in time, the thread's DebugUi will call the **DbgUiWaitStateChange** API. This API waits on the debug state change semaphore. When the semaphore becomes signaled, the DebugUi formats a DBGUI_WAIT_STATE_CHANGE message and sends the message to the Dbg server.

- Upon receipt of the DBGUI_WAIT_STATE_CHANGE message, the Dbg server scans the state change database for the DebugUi. Finding a state change record, the Dbg server populates the DBGUI_WAIT_STATE_CHANGE message and replies to the DebugUi.

- The DebugUi returns from its call to **DbgUiWaitStateChange**. A state change type of **DbgExceptionStateChange** is is returned, along with the Client Id of the thread that originally caused the access violation. Using this information, the DebugUi can take appropriate action. This may include reading and writing the threads registers using **NtGetContextThread**/NtSetContextThread or reading and writing the processes virtual memory using **NtReadVirtualMemory**/NtWriteVirtualMemory.

- Once the DebugUi is done servicing the access violation, it can continue the thread's execution by calling **DbgUiContinue**. This API simply formats a DBGUI_CONTINUE message and sends it to the Dbg server.

- Upon receipt of the continue message, the Dbg server locates the target thread, assures that it is waiting to be continued, and that an appropriate continue status was passed. Dbg server then sends a continue datagram to the OS2SS. After this is complete, a reply is generated to the DebugUi which is then free to wait for further debug events.

- Upon receipt of the continue datagram by DbgSs DLL code runing in the OS2SS, the continue status is examined and appropriate callouts are made. The DLL code then generates a reply to the original DBGKM_EXCEPTION message.

- Upon receipt of this reply, the thread begins executing in the DbgKm component. All of the threads in its process are unfrozen and the thread returns to the exception dispatcher

2. Debug Architecture Partitioning

The NT OS/2 Debug Architecture partitions the work involved in debugging into a number of stages.

2.1 Debug Event Generation

Debug event generation is done in the Dbgk component of the NT OS/2 executive. For each debug event, the following occurs:

- The process in which the debug event is occuring in is located.
- If the process has a DebugPort, then all threads in the process are frozen.
- A DBGKM_APIMSG is formated to indicate the type of debug event. This message is sent through the processes DebugPort using **LpcRequestWaitReplyPort**.
- Upon receipt of the reply, all threads in the process are unfrozen.

Debug events are generated for a number of reasons:

- **Exception**. When a thread whose process has a DebugPort encounters an exception, a debug event is generated.
- **CreateThread**. When a thread begins executing in a process being debugged, a debug event is generated before the thread gets a chance to execute in kernel mode.
- **CreateProcess**. When the first thread in a process being debugged begins executing, a debug event is generated before the thread gets a chance to execute in kernel mode.
- **ExitThread**. When a thread exits in a process being debugged, a debug event is generated. This occurs as soon as the system detects that the thread is exiting and has updated the exit status for the thread.
- **ExitProcess**. When the last thread in a process being debugged exits, a debug event is generated. This occurs as soon as the the status of the process has been updated. Note that when the last thread in a process exits, an exit thread debug event is not generated.
- **MapSection**. When a process being debugged maps a view of a section backed by an image file, a debug event is generated.
- **UnMapSection**. When a process being debugged un-maps a view of a section backed by an image file, a debug event is generated.

2.1.1 Event Generation Message Formats

Event generation messages are always sent in the context of the thread reporting the event; therefore, the client id stored in the message header can be used to determine the thread reporting the event. Event generation messages consist of the following standard header:

typedef struct **_DBGKM_APIMSG** {
    PORT_MESSAGE *h*;
    DBGKM_APINUMBER *ApiNumber*;
    NTSTATUS *ReturnedStatus*;
    union *u*;
} **DBGKM_APIMSG**, ***PDBGKM_APIMSG**;

DBGKM_APIMSG Structure:

h —Supplies the standard LPC port message. The ClientId field of this structure supplies the client id of the thread reporting the debug event. The message type field (h.u2.s2.Type) is LPC_DEBUG_EVENT.

ApiNumber —Supplies the *ApiNumber* for this message. The *ApiNumber* is used to indicate the type of event being generated.

ReturnedStatus —Returns the continuation status for the debug event.

u —Supplies the type specific event information.

2.1.1.1 Exception

If the *ApiNumber* is **DbgKmExceptionApi**, then *u*.Exception supplies a DBGKM_EXCEPTION message. The format of this message follows:

typedef struct **_DBGKM_EXCEPTION** {
    EXCEPTION_RECORD *ExceptionRecord*;
    BOOLEAN *FirstChance*;
} **DBGKM_EXCEPTION**, ***PDBGKM_EXCEPTION**;

DBGKM_EXCEPTION Structure:

ExceptionRecord —Supplies the exception record describing this exception.

FirstChance —Supplies a variable that if TRUE, indicates that this is the first time this debug event is being reported for this thread.

2.1.1.2 CreateThread

If the *ApiNumber* is **DbgKmCreateThreadApi**, then *u*.CreateThread supplies a DBGKM_CREATE_THREAD message. The format of this message follows:

typedef struct **_DBGKM_CREATE_THREAD** {
    ULONG *SubSystemKey*;
    PVOID *StartAddress*;
} **DBGKM_CREATE_THREAD**, ***PDBGKM_CREATE_THREAD**;

DBGKM_CREATE_THREAD Structure:

SubSystemKey —This field is initialized to 0.

StartAddress —Supplies the initial starting address for the thread. This is really advisory, since anyone with THREAD_SET_CONTEXT access to the thread may change this and superseed the value of this field.

2.1.1.3 CreateProcess

If the *ApiNumber* is **DbgKmCreateProcessApi**, then *u*.CreateProcess supplies a DBGKM_CREATE_PROCESS message. The format of this message follows:

typedef struct **_DBGKM_CREATE_PROCESS** {
    ULONG *SubSystemKey*;
    HANDLE *Section*;
    DBGKM_CREATE_THREAD *InitialThread*;
} **DBGKM_CREATE_PROCESS**, ***PDBGKM_CREATE_PROCESS**;

DBGKM_CREATE_PROCESS Structure:

SubSystemKey —This field is initialized to 0.

Section —Supplies a handle to the section object that describes the initial address space of the process. If this field is NULL, then no handle exists. The handle is valid in the sending processes handle table.

InitialThread —Supplies a description of the first thread to execute in the process.

2.1.1.4 ExitThread

If the *ApiNumber* is **DbgKmExitThreadApi**, then *u*.ExitThread supplies a DBGKM_EXIT_THREAD message. The format of this message follows:

typedef struct **_DBGKM_EXIT_THREAD** {
    NTSTATUS *ExitStatus*;
} **DBGKM_EXIT_THREAD**, ***PDBGKM_EXIT_THREAD**;

DBGKM_EXIT_THREAD Structure:

ExitStatus —Supplies the exit status of the exiting thread.

2.1.1.5 ExitProcess

If the *ApiNumber* is **DbgKmExitProcessApi**, then *u*.ExitProcess supplies a DBGKM_EXIT_PROCESS message. The format of this message follows:

typedef struct **_DBGKM_EXIT_PROCESS** {
    NTSTATUS *ExitStatus*;
} **DBGKM_EXIT_PROCESS**, ***PDBGKM_EXIT_PROCESS**;

DBGKM_EXIT_PROCESS Structure:

ExitStatus —Supplies the exit status of the exiting process.

2.1.1.6 MapSection

If the *ApiNumber* is **DbgKmMapSectionApi**, then *u*.MapSection supplies a DBGKM_MAP_SECTION message. The format of this message follows:

typedef struct **_DBGKM_MAP_SECTION** {
    HANDLE *SectionHandle*;
    PVOID *BaseAddress*;
    ULONG *SectionOffset*;
    ULONG *ViewSize*;
} **DBGKM_MAP_SECTION**, ***PDBGKM_MAP_SECTION**;

DBGKM_MAP_SECTION Structure:

SectionHandle —Supplies a handle to the section mapped by the process. The handle is valid in the context of the sending process.

BaseAddress —Supplies the base address of where the section is mapped in the processes address space.

SectionOffset —Supplies the offset in the section where the processes mapped view begins.

ViewSize —Supplies the size of the mapped view.

2.1.1.7 UnMapSection

If the *ApiNumber* is **DbgKmUnMapSectionApi**, then *u*.UnMapSection supplies a DBGKM_UNMAP_SECTION message. The format of this message follows:

```
typedef struct _DBGKM_UNMAP_SECTION {
    PVOID BaseAddress;
} DBGKM_UNMAP_SECTION, *PDBGKM_UNMAP_SECTION;
```

DBGKM_UNMAP_SECTION Structure:

BaseAddress —Supplies the base address of where the section being un-mapped is in the processes address space.

2.2 Event Propagation

Event propagation occurs after a a thread receives a debug event message on a processes DebugPort. Upon receipt of the message, the thread adds any necessary information, and forwards the message to the Dbg server.

Event propagation occurs within an Emulation Subsystem. In order to minimize thread blocking in the subsystems, an asynchronous protocol is used to propagate debug events. The event propagation protocol occurs as follows:

- The event is generated. The thread generating the event reports the event using **LpcRequestWaitReplyPort** against its processes DebugPort. The thread remains blocked until a reply is received.

- The subsystem receives the debug event message. After processing the message, it determines whether or not to propagate the message to the Dbg server. If it does not propagate the message, then it must reply to the thread reporting the event.

- To propagate the message, a copy of the message is made, and is sent as a datagram to the Dbg server. After receiving the message, and receiving a "continue" from the controlling debugger user interface, the Dbg server sends a continue datagram back to the subsystem.

- A dedicated thread in the subsystem receives the continue datagram, locates the associated saved debug event message, and replies to the thread reporting the event.

2.2.1 Emulation Subsystem APIs for Event Propagation

2.2.1.1 DbgSsInitialize

An Emulation Subsystem initializes itself so that it can participate in the NT OS/2 debug architecture using the following API:

NTSTATUS
DbgSsInitialize(
IN HANDLE *KmReplyPort*,
IN PDBGSS_UI_LOOKUP *UiLookUpRoutine*,
IN PDBGSS_SUBSYSTEMKEY_LOOKUP *SubsystemKeyLookupRoutine* **OPTIONAL**,
IN PDBGSS_DBGKM_APIMSG_FILTER *KmApiMsgFilter* **OPTIONAL**
)

Parameters:

KmReplyPort —Supplies a handle to the port that the subsystem receives DbgKm API messages on.

UiLookupRoutine —Supplies the address of a function that will be called upon receipt of a process creation message. The purpose of this function is to identify the client id of the debug user interface controlling the process.

SubsystemKeyLookupRoutine —Supplies the address of a function that will be called upon receipt of process creation and thread creation messages. The purpose of this function is to allow a subsystem to correlate a key value with a given process or thread.

KmApiMsgFilter —Supplies the address of a function that will be called upon receipt of a DbgKm Api message. This function can take any action. If it returns any value other than DBG_CONTINUE, **DbgSsHandleKmApiMsg** will not process the message. This function is called before any other call outs occur.

Return Value:

SUCCESS() —Initialization complete.

!SUCCESS() —Failure occured while connecting to the Dbg server

This function is called by a subsystem to initialize portions of the debug subsystem dll. The main purpose of this function is to set up callouts that are needed in order to use **DbgSsHandleKmApiMsg**, and to connect to the Dbg server.

2.2.1.2 UiLookupRoutine

The *UiLookupRoutine* is called during the propagation of create process debug events. Its function is to locate the client id of the debugger user interface that controls the process whose creation is being reported.

NTSTATUS
(***PDBGSS_UI_LOOKUP**)(
IN PCLIENT_ID *AppClientId*,
OUT PCLIENT_ID *DebugUiClientId*
)

Parameters:

AppClientId —Supplies the client id of the application thread reporting the create process debug event.

DebugUiClientId —Returns the client id of the debugger user interface that controls the application process.

Return Value:

STATUS_SUCCESS —The application is being debugged, and the client id of its debugger user interface has been returned.

!SUCCESS() —The application is not known as being debugged. The create process debug event will not be propagated to the debug server. A reply is generated and sent to the thread reporting the debug event.

2.2.1.3 SubsystemKeyLookupRoutine

The *SubsystemKeyLookupRoutine* is called during the propagation of create process and create thread debug events. Its function is to allow a subsystem to associate a key value with the process or thread whose creation is being reported. Examples of this are OS/2 might want to associate a TID with each thread, and a PID with each process. The subsystem key value is informational only.

NTSTATUS
(*PDBGSS_SUBSYSTEMKEY_LOOKUP)(
IN PCLIENT_ID *AppClientId*,
OUT PULONG *SubsystemKey*,
IN BOOLEAN *ProcessKey*
)

Parameters:

AppClientId —Supplies the client id of the application thread reporting the create process or create thread debug event.

SubsystemKey —Returns the subsystem key value to associate with the new process or thread.

ProcessKey —Supplies a flag which if TRUE indicates that a subsystem key for the process is needed; otherwise, a subsystem key for the thread is needed. Note that during the propagation of a create process debug event, this function is called twice. It is called once with *ProcessKey* set to TRUE, and once with *ProcessKey* set to FALSE.

Return Value:

STATUS_SUCCESS —A subsystem key was found and returned.

!SUCCESS() —A subsystem key was not located. This does not affect the propagation of the create process or create thread debug events. The *SubSystemKey* fields in the propagated messges will remain 0.

2.2.1.4 KmApiMsgFilter

The *KmApiMsgFilter* routine is called prior to debug event message propagation. The main purpose of this API is to allow a subsystem an opportunity to monitor debug events and to cancel the propagation of events.

NTSTATUS
(*PDBGSS_DBGKM_APIMSG_FILTER)(
IN OUT PDBGKM_APIMSG *ApiMsg*
)

Parameters:

ApiMsg —Supplies the DBGKM_APIMSG that is about to be propagated.

Return Value:

DBG_CONTINUE —Message propagation will continue. Note that since this callout occurs before the other callouts, event propagation can still be cancelled (by the *UiLookupRoutine*).

Others —The message will not be propagated. No reply is generated for the debug event.

2.2.1.5 DbgSsHandleKmApiMsg

The **DbgSsHandleKmApiMsg** is called by a subsystem whenever a debug event message arrives. This is typically done in the subsystem's main API loop whenever a message arrives whose type is LPC_DEBUG_EVENT.

VOID
DbgSsHandleKmApiMsg(
IN PDBGKM_APIMSG *ApiMsg*
)

Parameters:

ApiMsg —Supplies the debug event message to propagate to the debug server.

A number of callouts are performed prior to propagating the message:

- For all messages, the *KmApiMsgFilter* is called (if it was supplied during **DbgSsInitialize**). If this returns anything other that DBG_CONTINUE, the message is not propagated by this function. The caller is responsible for event propagation, and for replying to the thread reporting the debug event.

- For create process messages, the *UiLookupRoutine* is called. If a success code is returned than message is propagated. Otherwise, a reply is generated to the thread reporting the debug event.

- For create process and create thread messages, *SubsystemKeyLookupRoutine* is called. Failure does not effect propagation. It simply inhibits the update of the messages *SubSystemKey* field.

2.2.2 Event Propagation Message Formats

Event propagation messages are sent as datagrams to the Dbg server. Event propagation messages consist of the following standard header:

typedef struct **_DBGSS_APIMSG** {
    PORT_MESSAGE *h*;
    DBGKM_APINUMBER *ApiNumber*;
    NTSTATUS *ReturnedStatus*;
    CLIENT_ID *AppClientId*;
    PVOID *ContinueKey*;
    union *u*.
} **DBGSS_APIMSG**, ***PDBGSS_APIMSG**;

DBGSS_APIMSG Structure:

h —Supplies the standard LPC port message.

ApiNumber —Supplies the *ApiNumber* for this message. The *ApiNumber* is used to indicate the type of event being propagated.

ReturnedStatus —Used to store the continuation status for the event.

AppClientId —Supplies the client id of the application thread reporting the debug event. This comes directly from the header of the associated DBGKM_APIMSG.

ContinueKey —Supplies the continue key, that must be returned from the Dbg server in order to cause a reply to be generated for the thread that is reporting the debug event.

u —Supplies the type specific event propagation information.

2.2.2.1 Exception

If the *ApiNumber* is **DbgSsExceptionApi**, then *u*.Exception supplies a DBGKM_EXCEPTION message. The message contains the same information as it did at the time the event was generated.

2.2.2.2 CreateThread

If the *ApiNumber* is **DbgSsCreateThreadApi**, then *u*.CreateThread supplies a DBGKM_CREATE_THREAD message. The message contains the same information as it did at the time the event was generated. If a *SubsystemKeyLookupRoutine* was called and returned success, then the *SubSystemKey* field is modified appropriately.

2.2.2.3 CreateProcess

If the *ApiNumber* is **DbgSsCreateProcessApi**, then *u*.CreateProcess supplies a DBGSS_CREATE_PROCESS message. The format of this message follows:

typedef struct **_DBGSS_CREATE_PROCESS** {
    CLIENT_ID *DebugUiClientId*;
    DBGKM_CREATE_PROCESS *NewProcess*;
} **DBGSS_CREATE_PROCESS**, ***PDBGSS_CREATE_PROCESS**;

DBGSS_CREATE_PROCESS Structure:

DebugUiClientId —Supplies the client id of the processes debugger user interface.

NewProcess —Supplies the original contents of the DBGKM_CREATE_PROCESS message at the time the event was generated. If a *SubsystemKeyLookupRoutine* was called and returned success, then the *SubSystemKey* field is modified appropriately.

2.2.2.4 ExitThread

If the *ApiNumber* is **DbgSsExitThreadApi**, then *u*.ExitThread supplies a DBGKM_EXIT_THREAD message. The message contains the same information as it did at the time the event was generated.

2.2.2.5 ExitProcess

If the *ApiNumber* is **DbgSsExitProcessApi**, then *u*.ExitProcess supplies a DBGKM_EXIT_PROCESS message. The message contains the same information as it did at the time the event was generated.

2.2.2.6 MapSection

If the *ApiNumber* is **DbgSsMapSectionApi**, then *u*.MapSection supplies a DBGKM_MAP_SECTION message. The message contains the same information as it did at the time the event was generated.

2.2.2.7 UnMapSection

If the *ApiNumber* is **DbgSsUnMapSectionApi**, then *u*.UnMapSection supplies a DBGKM_UNMAP_SECTION message. The message contains the same information as it did at the time the event was generated.

2.3 Coordinate Debugger and Debuggee

The purpose of the Dbg server is to coordinate debug events occuring in the applications being debugged (debugee) with the requests for event notification coming from the applications debug user interface. In order to facilitate this type of coordination, the Dbg server maintains a set of data structures that bind a debugger with its debuggess, and that bind debuggees with their controlling subsystem.

The data structures used to do the bindings are created and modified based on the receipt of propagated debug event messages, connections to the Dbg server, and the receipt of wait and continue messages from debuggers.

The following sections describe the data structures maintained by the Dbg server, and the actions that causes the data structures to be created and modified.

2.3.1 Dbg Server Data Structures

2.3.1.1 Subsystem Structure

A subsystem structure exists for each subsystem connected to the Dbg server. A subsystem connects to the Dbg server as part of **DbgSsInitialize**. The connection port used during the connection has a security descriptor that limits access to only those processes that form part of the NT OS/2 TCB.

```
typedef struct _DBGP_SUBSYSTEM {
        CLIENT_ID SubsystemClientId;
        HANDLE CommunicationPort;
        HANDLE SubsystemProcessHandle;
} DBGP_SUBSYSTEM, *PDBGP_SUBSYSTEM;
```

DBGP_SUBSYSTEM Structure:

SubsystemClientId —Contains the client id of the subsystem thread that initially connects to the Dbg server.

CommunicationPort —Contains a handle to the communication port used to send continue datagrams back to the subsystem.

SubsystemProcessHandle —Contains a handle to the subsystem process. This handle has PROCESS_DUP_HANDLE access to the process.

2.3.1.2 User Interface Structure

A user interface structure is maintained for each debugger user interface (DebugUi) connected to the Dbg server. Debugger user interface's connect to the Dbg server as part of their initialization process. The connection must be made before the user interface starts any applications that need to be debugged.

The user interface structure is the key coordination data structure. All of the application processes and threads controlled by a user interface are linked off of the data structure.

typedef struct **_DBGP_USER_INTERFACE** {
    CLIENT_ID *DebugUiClientId*;
    HANDLE *CommunicationPort*;
    HANDLE *DebugUiProcess*;
    HANDLE *StateChangeSemaphore*;
    RTL_CRITICAL_SECTION *UserInterfaceLock*;
    LIST_ENTRY *AppProcessListHead*;
    LIST_ENTRY *HashTableLinks*;
} **DBGP_USER_INTERFACE**, ***PDBGP_USER_INTERFACE**;

DBGP_USER_INTERFACE Structure:

DebugUiClientId —Contains the client id of the thread that connects to the Dbg server. During the processing of a propagated create process debug event, the *DebugUiClientId* field of a DBGSS_CREATE_PROCESS structure is matched against this field. Once a match is found, the process is bound to the user interface.

CommunicationPort —Contains a handle to the communication port used to reply back to requests from the user interface.

DebugUiProcess —Contains a handle to the user interface process. The handle has PROCESS_DUP_HANDLE access to the process. This handle is used to duplicate object handles into and out of the user interace.

StateChangeSemaphore —Contains a handle to a semaphore shared between the Dbg server and the user interface. The semaphore is signaled each time a propagated debug event is available to be picked up by the user interface. The user interface waits on this semaphore. When a wait is satisfied, the user interface can call into the Dbg server to receive notification of debug events.

UserInterfaceLock —Contains a critical section lock to gaurd the user interface and associated structures.

AppProcessListHead —Contains a list head where processes being debugged by the user interface are linked.

HashTableLinks —Contains a set of link words use to quickly locate a user interface by client id.

2.3.1.3 Application Process Structure

An application process structure is maintained for each process accepted by the Dbg server. For a process to be accepted, a process creation debug event must propagated to the Dbg server, and the *DebugUiClientId* field of the DBGSS_APIMSG must match the client id of a user interface connected to Dbg. Once this occurs, an application process structure is

typedef struct **_DBGP_APP_PROCESS** {
    LIST_ENTRY *AppThreadListHead*;
    LIST_ENTRY *AppLinks*;
    LIST_ENTRY *HashTableLinks*;
    CLIENT_ID *AppClientId*;
    PDBGP_USER_INTERFACE *UserInterface*;
    HANDLE *HandleToProcess*;
} **DBGP_APP_PROCESS**, ***PDBGP_APP_PROCESS**;

DBGP_APP_PROCESS Structure:

AppThreadListHead —Contains the list head for all application threads that form this process.

AppLinks —Contains the link words that are used to link the process to its user interface.

HashTableLinks —Contain a set of link words to quickly locate an application process by its UniqueProcess portion of its client id.

AppClientId —Contains the client id of the processes initial thread. Only the UniqueProcess portion of the client id is used.

UserInterface —Contains a pointer to the processes user interface.

HandleToProcess —Contains a handle to the application process. When the application process structure is initially created, a handle is created in the context of the Dbg server. The handle has PROCESS_VM_READ and PROCESS_VM_WRITE access. When the user interface waits for and receives notification of the new process debug event, the handle is duplicated into the user interface and closed in the Dbg server. This field is then modified so that the handle value is a handle in the context of the user interface. When the user interface issues a continue after an exit process debug event, the handle to the process is closed and is no longer available to the user interface.

2.3.1.4 Application Thread Structure

An application thread structure is maintained for each thread accepted by the Dbg server. For a thread to be accepted, a thread creation debug event must propagated to the Dbg server, and the process that the thread is part of must have been previously accepted by Dbg.

typedef struct **_DBGP_APP_THREAD** {
    LIST_ENTRY *AppLinks*;
    LIST_ENTRY *HashTableLinks*;
    CLIENT_ID *AppClientId*;
    DBG_STATE *CurrentState*;
    DBG_STATE *ContinueState*;
    PDBGP_APP_PROCESS *AppProcess*;
    PDBGP_USER_INTERFACE *UserInterface*;
    HANDLE *HandleToThread*;
    PDBGP_SUBSYSTEM *Subsystem*;
    DBGSS_APIMSG *LastSsApiMsg*;
} **DBGP_APP_THREAD**, ***PDBGP_APP_THREAD**;

DBGP_APP_THREAD Structure:

AppLinks —Contains link words that link an application thread to its process.

HashTableLinks —Contain a set of link words to quickly locate an application thread by its client id.

AppClientId —Contains the client id of the thread.

CurrentState —Contains the Dbg server maintained state for the thread. The thread can be in three state clasess. If a debug event has been propagated to and has been accepted by the Dbg server, the state class is "state change available". Once a user interface has received notification of the state change, the class becomes "continue pending". When a user interface issues a continue to a thread whose state class is "continue pending", the threads state class returns to "idle".

ContinueState —Contains the saved Dbg server state for the thread at the time the thread transitions from "state change available" to "continue pending".

AppProcess —Contains a pointer to the threads process.

UserInterface —Contains a pointer to the threads user interface.

HandleToThread —Contains a handle to the application thread. When the application thread structure is initially created, a handle is created in the context of the Dbg server. The handle has THREAD_GET_CONTEXT and THREAD_SET_CONTEXT access. When the user interface waits for and receives notification of the new thread debug event, the handle is duplicated into the user interface and closed in the Dbg server. This field is then modified so that the handle value is a handle in the context of the user interface. When the user interface issues a continue after an exit thread debug event, or after an exit process debug event, the handle to the thread is closed and is no longer available to the user interface.

Subsystem —Contains a pointer to the thread's subsystem. This is used to locate the subsystem to send a continue datagram to when the thread's user interface issues a continue.

LastSsApiMsg —This field contains the last DBGSS_APIMSG for the thread. This message is valid while a thread is in the "state change available" state class. Portions of this message are made available to the user interface when it receives notification of the occurance of a debug event.

2.3.2 Dbg Server Responses to Debug Event Propagation

The Dbg server responds to propagated debug events by creating or modifying its user interface, application process, or application thread data structures.

Debug events are are propagated to the Dbg server by a subsystem that is connected to Dbg and is identified by its subsystem structure. Upon receipt of a propagated debug event message, Dbg determines whether or not to accept the message. If the message is accepted, then the message is captured into the appropriate application thread structure, the thread's state is changed to the "state change available" class, and the appropriate user interface's *StateChangeSemaphore* is signaled.

The following sections describe the actions taken by the Dbg server upon receipt of a propagated debug event message:

2.3.2.1 Exception

The application thread structure for the specified thread is located. If the thread can not be found, or if the thread is not known to the Dbg server, or if the thread is not "idle", the message is not accepted and an appropriate continue datagram is sent to the appropriate subsystem; otherwise, the exception message is accepted and the following occurs:

- The *CurrentState* and *ContinueState* fields of thread are modified to either **DbgBreakpointStateChange**, **DbgSingleStepStateChange**, or DbgExceptionState change base on the **ExceptionCode** field of the *ExceptionRecord*.

2.3.2.2 CreateThread

The application process structure that the thread is part of is located. If the process can not be found , the message is not accepted and an appropriate continue datagram is sent to the appropriate subsystem; otherwise, the create thread message is accepted and the following occurs:

- An application thread structure for the thread is allocated.
- The *CurrentState* and *ContinueState* fields are initialized to **DbgCreateThreadStateChange**.
- The *AppProcess* field is set to point to the thread's process, the *UserInterface* field is initialized to point to the thread's user interface, the *AppClientId* field is initialized, and the thread is linked to its process.

- A handle to the thread is created in the context of the Dbg server. If this operation succeeds, then the *HandleToThread* field is initialized to the value of the handle; otherwise, it is initialized to NULL.

2.3.2.3 CreateProcess

The user interface whose client id is specified in the message is located. If the user interface can not be located (the user interface has not connected to Dbg), the message is not accepted and an appropriate continue datagram is sent to the appropriate subsystem; otherwise, the create process message is accepted and the following occurs:

- An application process structure for the process is allocated.
- The *UserInterface* field is initialized to point to the process' user interface, the *AppClientId*.UniqueProcess field is initialized.
- The process is linked to its user interface.
- A handle to the process is created in the context of the Dbg server. If this operation succeeds, then the *HandleToProcess* field is initialized to the value of the handle; otherwise, it is initialized to NULL.
- An application thread structure for the thread described in the create process message is allocated.
- The *CurrentState* and *ContinueState* fields are initialized to **DbgCreateProcessStateChange**.
- The *AppProcess* field is set to point to the thread's process, the *UserInterface* field is initialized to point to the thread's user interface, the *AppClientId* field is initialized, and the thread is linked to its process.
- A handle to the thread is created in the context of the Dbg server. If this operation succeeds, then the *HandleToThread* field is initialized to the value of the handle; otherwise, it is initialized to NULL.

2.3.2.4 ExitThread

The application thread structure for the specified thread is located. If the thread can not be found, or if the thread is not known to the Dbg server, or if the thread is not "idle", the message is not accepted and an appropriate continue datagram is sent to the appropriate subsystem; otherwise, the exit thread message is accepted and the following occurs:

- The *CurrentState* and *ContinueState* fields of thread are set to **DbgExitThreadStateChange**.

2.3.2.5 ExitProcess

The application thread structure for the specified thread is located. If the thread can not be found, or if the thread is not known to the Dbg server, or if the thread is not "idle", the message is not accepted and an appropriate continue datagram is sent to the appropriate subsystem; otherwise, the exit process message is accepted and the following occurs:

- The *CurrentState* and *ContinueState* fields of thread are set to **DbgExitProcessStateChange**.

2.3.2.6 MapSection

The application thread structure for the specified thread is located. If the thread can not be found, or if the thread is not known to the Dbg server, or if the thread is not "idle", the message is not accepted and an appropriate continue datagram is sent to the appropriate subsystem; otherwise, the map section message is accepted and the following occurs:

- The *CurrentState* and *ContinueState* fields of thread are set to **DbgMapSectionStateChange**.

2.3.2.7 UnMapSection

The application thread structure for the specified thread is located. If the thread can not be found, or if the thread is not known to the Dbg server, or if the thread is not "idle", the message is not accepted and an appropriate continue datagram is sent to the appropriate subsystem; otherwise, the un-map section message is accepted and the following occurs:

- The *CurrentState* and *ContinueState* fields of thread are set to **DbgUnMapSectionStateChange**.

2.4 User Interface Interactions with the Dbg Server

A debug user interface has three main interactions with the Dbg server.

- Connecting to the Dbg server
- Waiting for debug event state changes to occur
- Continuing an application thread

2.4.1 DbgUiConnectToDbg

A user interface can connect to the Dbg server using **DbgUiConnectToDbg**.

NTSTATUS
DbgUiConnectToDbg(**VOID**)

Return Value:

SUCCESS() —A connection between the user interface and the Dbg server has been made. The DbgUi dll has been initialized.

!SUCCESS() —The connection to Dbg did not occur. The user interface can not use services provided by the Dbg server.

This routine makes a connection between the calling user interface and the Dbg server. If the routine is successful, a communications port is created to link the user interface with the Dbg server. A user interface data structure is created and initialized in the Dbg server. A shared state change semaphore is created between the Dbg server and the user interface. The Dbg server is granted SEMAPHORE_ALL_ACCESS to the semaphore. This allows it to signal the semaphore at the appropriate times. A handle to the semaphore is duplicated to the user interface. The handle is granted SYNCHRONIZE access to the semaphore. This allows the user interface an opportunity to wait on the semaphore. The semaphore becomes signaled when a propagated debug event is available which transitions one of the user interface's threads into the "state change available" state.

2.4.2 DbgUiWaitStateChange

A user interface can wait for a state change to occur in one of its threads using **DbgUiWaitStateChange**.

NTSTATUS
DbgUiWaitStateChange(
OUT PDBGUI_WAIT_STATE_CHANGE *StateChange*
)

Parameters:

StateChange —Supplies the address of state change record that will contain the state change information.

Return Value:

STATUS_USER_APC —A user mode APC occured which caused this call to abort without retreiving state change information.

STATUS_ALERTED —The thread was alerted while waiting for a state change to occur. No state change information was retreived.

DBG_NO_STATE_CHANGE —The state change semaphore was signaled, but the Dbg server has no state change information to return. This error can only happen if a user interface bypasses the DbgUi APIs and attempts to communicate directly with the Dbg server.

DBG_UNABLE_TO_PROVIDE_HANDLE —A state change occurred that required a handle to be duplicated into the user interface. For some reason, a handle could not be provided. All other portions of the state change reporting were successful.

STATUS_SUCCESS —A state change occured. Valid state change information was returned.

OTHERS —Refer to object management error codes.

This function causes the calling user interface to wait for a state change to occur in one of it's application threads. The wait is ALERTABLE. A state change occurs when an application thread changes its state to the "state change available" class. If a user interface makes a successful call to this function while one of its threads is in the "state change available" class, then the threads state is set to "continue pending", and a state change record is formatted and returned to the caller. Once a state change has been reported for a thread, its user interface is responsible for continuing the thread at the appropriate time.

2.4.2.1 State Change Record

A state change record has the following format:

typedef struct **_DBGUI_WAIT_STATE_CHANGE** {
    DBG_STATE *NewState*;
    CLIENT_ID *AppClientId*;
    union *StateInfo*;
} **DBGUI_WAIT_STATE_CHANGE**, ***PDBGUI_WAIT_STATE_CHANGE**;

DBGUI_WAIT_STATE_CHANGE Structure:

NewState —Supplies the new state of the thread reporting the state change.

AppClientId —Supplies the client id of the thread reporting the state change.

StateInfo —Supplies the per-state change type description that describes the state change.

2.4.2.1.1 DbgCreateThreadStateChange

The state change of **DbgCreateThreadStateChange** is reported whenever a state change is reported due to the propagation of a create thread debug event. The major side effect of this state change is that

the user interface is given a handle to the thread reporting the state change. The handle is granted THREAD_GET_CONTEXT and THREAD_SET_CONTEXT access to the thread. This allows the user interface to use NtReadContextThread and NtWriteContextThread to read and write the thread's registers.

StateInfo for this type of state change is as follows:

typedef struct **_DBGUI_CREATE_THREAD** {
    HANDLE *HandleToThread*;
    DBGKM_CREATE_THREAD *NewThread*;
} **DBGUI_CREATE_THREAD**, ***PDBGUI_CREATE_THREAD**;

DBGUI_CREATE_THREAD Structure:

HandleToThread —Supplies a handle to the thread identified by this state change. A value of NULL indicates that the handle is not valid and that an informational status code of DBG_UNABLE_TO_PROVIDE_HANDLE was returned.

NewThread —Supplies the description of the new thread as formatted by the subsystem during debug event propagation.

2.4.2.1.2 DbgCreateProcessStateChange

The state change of **DbgCreateProcessStateChange** is reported whenever a state change is reported due to the propagation of a create process debug event. The major side effects of this state change are:

- The user interface is given a handle to the process of the thread reporting the state change. The handle is granted PROCESS_VM_READ and PROCESS_VM_WRITE access to the process. This allows the user interface to use **NtReadVirtualMemory** and **NtWriteVirtualMemory** to read and write the processes virtual memory.

- The user interface is given a handle to the section that forms the initial address space for the process reporting the state change. The handle is granted SECTION_ALL_ACCESS access to the section. This allows the user interface to map a view of the section to locate the symbol table and other section information.

- The user interface is given a handle to the thread reporting the state change. The handle is granted THREAD_GET_CONTEXT and THREAD_SET_CONTEXT access to the thread. This allows the user interface to use NtReadContextThread and NtWriteContextThread to read and write the thread's registers.

typedef struct **_DBGUI_CREATE_PROCESS** {
    HANDLE *HandleToProcess*;
    HANDLE *HandleToThread*;
    DBGKM_CREATE_PROCESS *NewProcess*;
} **DBGUI_CREATE_PROCESS**, ***PDBGUI_CREATE_PROCESS**;

DBGUI_CREATE_PROCESS Structure:

HandleToProcess —Supplies a handle to the process identified by this state change. A value of NULL indicates that the handle is not valid and that an informational status code of DBG_UNABLE_TO_PROVIDE_HANDLE was returned.

HandleToThread —Supplies a handle to the thread identified by this state change. A value of NULL indicates that the handle is not valid and that an informational status code of DBG_UNABLE_TO_PROVIDE_HANDLE was returned.

NewProcess —Supplies the description of the new process as formatted by the subsystem during debug event propagation. The **Section** field of this structure is modified to contain a handle that is valid in the user interfaces context. A value of NULL indicates that the handle is not valid and that an informational status code of DBG_UNABLE_TO_PROVIDE_HANDLE was returned.

2.4.2.1.3 DbgExitThreadStateChange

The state change of **DbgExitThreadStateChange** is reported whenever a state change is reported due to the propagation of an exit thread debug event. There are no side effects of this state change.

StateInfo for this type of state change is the same as that originally formatted by Dbgk during debug event generation.

2.4.2.1.4 DbgExitProcessStateChange

The state change of **DbgExitProcessStateChange** is reported whenever a state change is reported due to the propagation of an exit process debug event. There are no side effects of this state change.

StateInfo for this type of state change is the same as that originally formatted by Dbgk during debug event generation.

2.4.2.1.5 DbgExceptionStateChange

The state change of **DbgExceptionStateChange** is reported whenever a state change is reported due to the propagation of an exception debug event where the exception code is anything other than STATUS_BREAKPOINT or STATUS_SINGLE_STEP. There are no side effects of this state change.

StateInfo for this type of state change is the same as that originally formatted by Dbgk during debug event generation.

2.4.2.1.6 DbgBreakpointStateChange

The state change of **DbgBreakpointStateChange** is reported whenever a state change is reported due to the propagation of an exception debug event where the exception code is STATUS_BREAKPOINT . There are no side effects of this state change.

StateInfo for this type of state change is the same as that originally formatted by Dbgk during debug event generation.

2.4.2.1.7 DbgSingleStepStateChange

The state change of **DbgSingleStepStateChange** is reported whenever a state change is reported due to the propagation of an exception debug event where the exception code is STATUS_SINGLE_STEP . There are no side effects of this state change.

StateInfo for this type of state change is as that same as that originally formatted by Dbgk during debug event generation.

2.4.2.1.8 DbgMapSectionStateChange

The state change of **DbgMapSectionStateChange** is reported whenever a state change is reported due to the propagation of a map section debug event. The major side effects of this state change are:

- The user interface is given a handle to the section being mapped by reporting the state change. The handle is granted SECTION_ALL_ACCESS access to the section. This allows the user interface to map a view of the section to locate the symbol table and other section information.

2.4.2.1.9 DbgUnMapSectionStateChange

The state change of **DbgUnMapSectionStateChange** is reported whenever a state change is reported due to the propagation of an un-map section debug event. There are no side effects of this state change.

StateInfo for this type of state change is the same as that originally formatted by Dbgk during debug event generation.

2.4.3 DbgUiContinue

A user interface can continue a thread that previously reported a state change using **DbgUiContinue**.

NTSTATUS
DbgUiContinue(
    IN PCLIENT_ID *AppClientId*,
    IN NTSTATUS *ContinueStatus*
)

Parameters:

AppClientId —Supplies the address of the ClientId of the application thread being continued. This must be an application thread that previously notified the caller through **DbgUiWaitStateChange** but has not yet been continued.

ContinueStatus —Supplies the continuation status to the thread being continued. Valid values for this are DBG_EXCEPTION_HANDLED, DBG_EXCEPTION_NOT_HANDLED, DBG_TERMINATE_THREAD, DBG_TERMINATE_PROCESS, or DBG_CONTINUE.

Return Value:

STATUS_SUCCESS —Successful call to **DbgUiContinue**

STATUS_INVALID_CID —An invalid ClientId was specified for the *AppClientId*, or the specified Application was not waiting for a continue.

STATUS_INVALID_PARAMETER —An invalid continue status was specified.

Continuing an application thread has a number of side effects. In some cases data structures inside of the Dbg server are modified or event deallocated. This is all dependent upon the *ContinueState* of the thread being continued. A number of standard actions occur during a continue regardless of the thread's *ContinueState*:

- A check is made to ensure that the *ContinueStatus* is valid, that the thread is known to Dbg, and that the thread is in the "continue pending" state.
- Perform and *ContinueState* dependent side effects.
- Format a continue datagram and send it to the thread's subsystem. This is then picked up by the subsystem which uses the continue key to reply to the original DBGKM_APIMSG which generated the debug event. Once the reply is received the thread which genrated the original debug even can continue execution.

The following sections describe the *ContinueState* dependent side effects of **DbgExitThreadStateChange** and **DbgExitProcessStateChange** state changes. No other state change types have side effects.

2.4.3.1 DbgExitThreadStateChange

Continuing a thread whose continue state is **DbgExitThreadStateChange**, causes the Dbg server to deallocate its application thread structure. If a handle to the thread was successfully duplicated into the user interface, the handle is closed. Once a user interface continues a thread in this state, it can no longer read and write the thread's registers.

2.4.3.2 DbgExitProcessStateChange

Continuing a thread whose continue state is **DbgExitProcessStateChange**, causes the Dbg server to deallocate its application process structure. If a handle to the process was successfully duplicated into the user interface, the handle is closed. Since this also implies that an application thread has exited, the thread's application thread structure is deallocated. If a handle to the thread was successfully duplicated into the user interface, the handle is closed.

Once a user interface continues a thread in this state, it can no longer read and write the thread's registers, or read and write the processes virtual memory.