Portable Systems Group

Windows NT Session Management and Control

Author: *Mark Lucovsky*

Revision 1.9, January 7, 1990

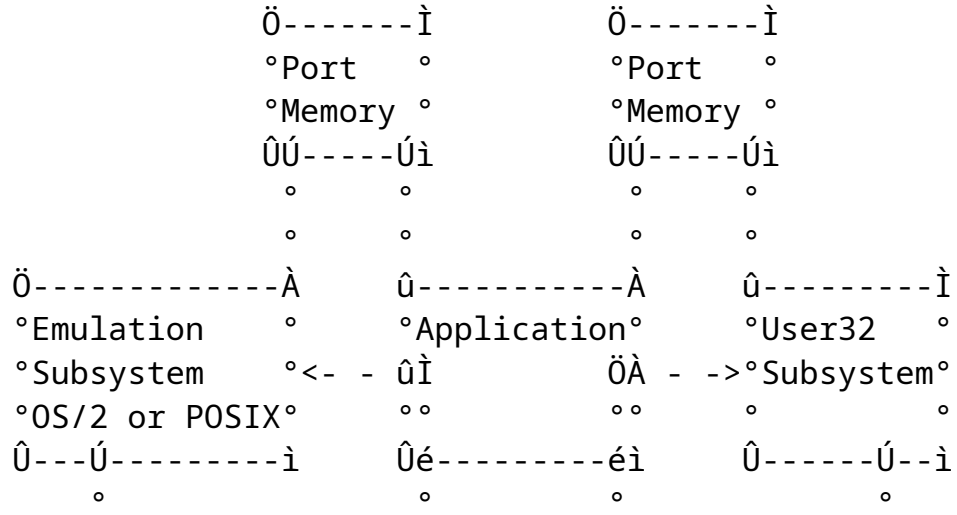
1. Introduction

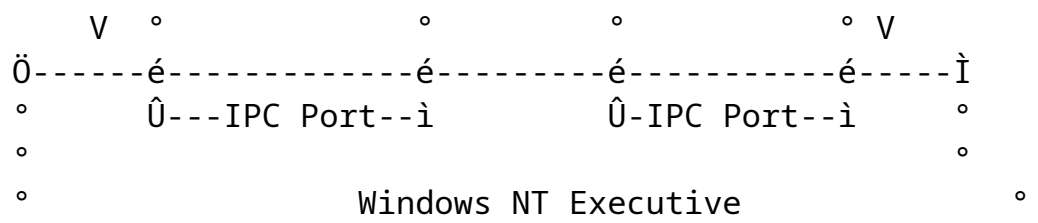
The **Windows NT** operating system is designed to support multiple concurrent application execution environments. The initial application execution environments that will be supported under **Windows NT** include **POSIX** (*IEEE Std 1003.1-1988*), and **32-Bit Cruiser OS/2**.

Users will see **Windows NT** as a system that lets them execute both **POSIX** and **OS/2** applications concurrently. There is no need to reboot the system to gain access to a particular execution environment.

Multiple concurrent application execution environments are made possible by implementing these environments as *Emulation Subsystems*. An *Emulation Subsystem* implements the APIs of a given operating system as a protected subsystem. Each application program image file header contains a description of the operating system environment that it has been designed to run in (e.g., **cmd.exe** is marked as an **OS/2** application and **ed** is marked as a **POSIX** application). During the process initialization of an application, an *LPC* connection is made between the application and the *Emulation Subsystem* that it has been designed to run with. Each system service API call that the application makes is translated into a *Local Procedure Call* (*LPC*) to the *Emulation Subsystem*. The subsystem implements the respective APIs using native **Windows NT** services.

The structure of an application program with respect to an *Emulation Subsystem* and the Native **Windows NT** System Services is depicted below.





1.1 NT Sessions

Windows NT provides a mechanism that allows an application in one environment to execute an application designed to run in another environment. For example, the **OS/2** command line interpreter **cmd.exe** can start the **POSIX** editor **ed** as follows.

- **cmd.exe**, an **OS/2** application calls **DosExecPgm** passing it the program name **ed**.- The **OS/2** subsystem creates a process ready to execute the **ed** program.
- After creating the process, the image type is examined.
- Since the image type indicates that it is not an **OS/2** application, the **OS/2** subsystem issues an *LPC* to *Sm* asking it to forward the process off to an appropriate *Emulation Subsystem*. *Sm* exports an API named **SmCreateForeignSession** that performs this function.
- *Sm* examines the image type passed as part of the **SmCreateForeignSession** call. The image type indicates that **ed** is a **POSIX** application.
- *Sm* issues an *LPC* to the **POSIX** subsystem passing it the process (originally created by the **OS/2** subsystem). Each *Emulation Subsystem* exports an API named **SbCreateSession** that performs this function.
- When the **ed** application terminates, the **POSIX** subsystem issues an *LPC* to *Sm* indicating that the process has completed with the specified termination status. *Sm* exports an API named **SmSessionComplete** that performs this function.
- Upon receipt of the call, *Sm* issues an *LPC* to the **OS/2** subsystem indicating that **ed** has terminated with the specified termination

status. Each *Emulation Subsystem* exports an API named **SbForeignSessionComplete** that performs this function.

In addition to starting an application in a different environment, **Windows NT** allows an application in one environment to pass information through a pipe stream to a process in another environment. The *Pipe Stream Subsystem* (*Piper*) exports a set of APIs used by *Emulation Subsystems* that make this possible.

1.2 NT Logon Sessions

To tie all related NT sessions together, a *logon session* is used. A logon session serves as a parent to all sessions related to a single logon.

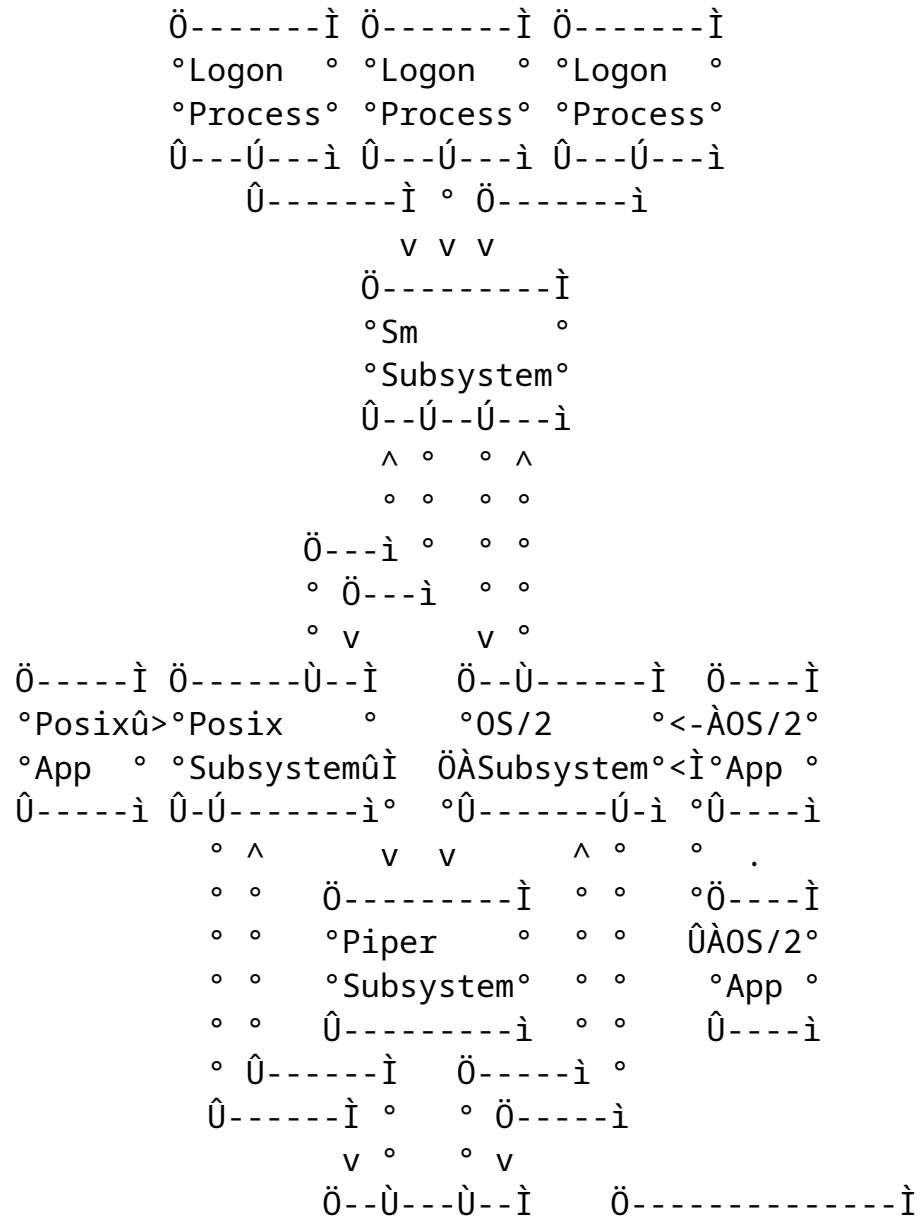
Associated with a logon session, and all the sessions related to it, is an object directory refered to as the *Logon Object Directory*. This object directory may be used to house objects related to processes related to all sessions of the logon session. The name of the logon object directory may be obtained using the **SmGetLogonObjectDirectory()** service.

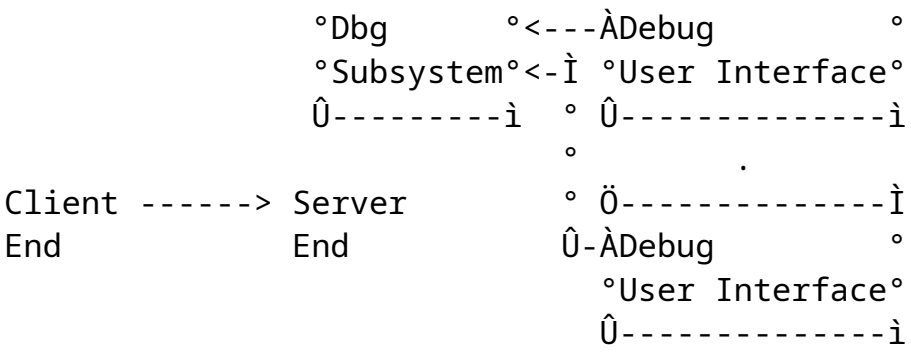
Throughout this document, the term *session* typically referes to an NT session. When a higher level *logon session* is being refered to, it will explicitly be called out as a logon session.

1.3 Windows NT System Structure

Before going any further, the following diagram is presented to show the overall structure of the subsystems and system processes that implement the session management and control portion of the **Windows NT** operating system.

Windows NT System Structure





The above diagram shows the structure of a **Windows NT** system. Most of the structure is static and is created at system boot time. The purpose of each component is described below.

Logon Processes - A logon process is created for each class of devices that can accept and process logon requests. Each logon process exists as a client process served by *Sm*. The *LPC* connection between a logon process and *Sm* is trusted and relatively static (created when each logon process initializes). A logon process is responsible for detecting logon requests from the devices it manages, authenticating the user (using the Local Security Authority), and calling *Sm* to activate the logon shell for the newly logged on user.

Sm Subsystem - The *Sm* subsystem is created during system initialization as the initial user mode process. It is responsible for building the structure presented in the above diagram. After the structure is built, *Sm* acts as the system session manager. In this role it is responsible for activating new logon shell programs and for fielding process creation requests from the various *Emulation Subsystem* and forwarding them on to the appropriate *Emulation Subsystem*.

This occurs when a subsystem is instructed to execute a program image, and the image file header describes an image designed to run in a different environment. *Sm* acts as a server to both logon processes and *Emulation Subsystems*.

As a server, *Sm* exports the following APIs over a trusted *LPC* connection between an *Emulation Subsystem* and itself:

- o - **SmConnectToSm** - Called by an *Emulation Subsystem* to create an *LPC* connection to *Sm*.

- o - **SmCreateForeignSession** - Called by an *Emulation Subsystem* when it detects an image file designed to execute in a different environment.

- **SmTerminateForeignSession** - Called by an *Emulation Subsystem* when it wants to terminate a session that it has asked *Sm* to create.

- **SmSessionComplete** - Called by an *Emulation Subsystem* when a session it has been asked to create completes.

- **SmGetLogonObjectDirectory** - Called by an *Emulation Subsystem* to determine the logon object directory associated with a session.

As a server, *Sm* exports the following APIs over a trusted *LPC* connection between a *Logon Process* and itself:

- **SmConnectToSm** - Called by an *Logon Process* to create an *LPC* connection to *Sm*.

- **SmRegisterLogonProcess** - Called by a *Logon Process* to identify itself as a logon process. This is called after connecting to *Sm* using **SmConnectToSm**.

- **SmExecLogonShell** - Called by a *Logon Process* to activate a user interface shell program for a new interactive logon session. This is used after the user has been authenticated, and a token obtained from the Local Security Authority.

Sm acts as a client of the *Emulation Subsystems*. As a client, *Sm* makes the following API calls over trusted *LPC* connections between an *Emulation Subsystem* and itself:

- **SbCreateSession** - *Sm* calls this API to implement a portion of **SmCreateForeignSession**. After examining the image type, *Sm* directs this call to the appropriate *Emulation Subsystem*.

- **SbTerminateSession** - *Sm* calls this API to implement a portion of **SmTerminateForeignSession**. After locating the *Emulation Subsystem* responsible for the specified session ID, *Sm* makes this call to the *Emulation Subsystem*.

- **SbForeignSessionComplete** - *Sm* calls this API to implement a portion of **SmSessionComplete**. After locating the *Emulation Subsystem* responsible for the specified session ID, *Sm* makes this call to the *Emulation Subsystem*.

Emulation Subsystems - *Emulation Subsystems* implement the operating system service APIs for a given operating system environment. In this role, *Emulation Subsystems* act as "system service servers" exporting system service APIs between themselves and the applications that run in a particular environment. The *LPC* connections between an application and its *Emulation Subsystem* are not trusted. When an *Emulation Subsystem* is called it can determine if it created the calling thread and can fail the call if appropriate.

Emulation Subsystems maintain connections to other subsystems as well. These connections are static connections created at system initialization time and are trusted. Each *Emulation Subsystem* maintains the following static connections:

- A pair of connections is maintained between each *Emulation Subsystem* and *Sm*. One connection is used when the *Emulation Subsystem* is acting as a server to export the **Sb...** APIs to *Sm*. The other connection is used when the *Emulation Subsystem* is acting as a client calling the **Sm...** APIs.

- A single connection is maintained between each *Emulation Subsystem* and *Piper*. This connection allows the subsystem to pass pipe stream input and output between itself and another *Emulation Subsystem*. The *Emulation Subsystem* is responsible for determining when I/O needs to be serviced using APIs available over this connection. The **Windows NT** I/O system is not involved in this decision.

- A pair of connections is maintained between each *Emulation Subsystem* and the *Debugger Subsystem* (*Dbg*). One connection is used when the *Emulation Subsystem* is acting as a server to export the **SbDebugSupport** API to *Dbg*. This API lets *Dbg* read and write the memory and context

associated with the specified thread, and to control the execution (start, stop, terminate) of the specified thread. The other connection is used by the *Emulation Subsystem* to notify *Dbg* of significant events occuring in a "debugged" thread or process (e.g., encountering an exception, process or thread creation, process or thread termination).

- o - A pair of implicit connections are maintained between each *Emulation Subsystem* and the **Windows NT** executive. These connections can act as the "*DebugPort* and *ExceptionPort*" values specified in a call to **NtCreateProcess**. Upon receipt of an exception, the **Windows NT** executive examines the process of the thread in which the exception occured. If the process was created with either a *DebugPort* or an *ExceptionPort*, then the *Emulation Subsystem* is notified of the exception over this connection.

Piper Subsystem - *Piper* is implemented as a server subsystem that views *Emulation Subsystems* as its clients. *Piper* only maintains trusted *LPC* connections between itself and the *Emulation Subsystems*. *Piper* is responsible for maintaining read/write data streams. *Piper* exports the following APIs:

- o - **PiperCreatePipe** - This API causes the *Piper* to create a pipe stream accessible to processes in the specified sessions. The data in the stream is only available by having the process' *Emulation Subsystem* call *Piper*.

- o - **PiperJoinPipe** - This API causes the *Piper* to bind to a pipe stream so that data can flow over the pipe.

- o - **PiperLeavePipe** - This API causes the *Piper* to close one end of a pipe stream. Once both ends of a pipe stream are closed, the pipe and any remaining data become inaccesible.

- o - **PiperReadPipe** - This API causes the *Piper* to return data stored in the pipe stream making room for new data.

- **PiperWritePipe** - This API causes the *Piper* to store data in the specified pipe stream.

Dbg Subsystem - The *Dbg Subsystem* implements the machine dependent facilities needed to debug an application thread. For more information on the *Dbg Subsystem*, refer to the **Windows NT Debug Architecture** document.

2. General Sm Services

The *Sm* has several classes of client, and provides services tailored to each class. The services that are used by more than one class of client are:

SmConnectToSm
SmGetLogonObjectDirectory

These services are described in the following subsections.

2.1 SmConnectToSm

NTSTATUS
SmConnectToSm(
IN PSTRING *SbApiPortName* **OPTIONAL**,
IN HANDLE *SbApiPort* **OPTIONAL**,
OUT PHANDLE *SmApiPort*
);

Parameters:

SbApiPortName - An optional string that if supplied specifies the name of a connection port that *Sm* will use to connect back to the *Emulation Subsystem*. This parameter is only used by *Emulation Subsystems* that are known to *Sm*.

SbApiPort - A optional handle that if supplied specifies a handle to a port named by the *SbApiPortName* parameter. This parameter is only used by *Emulation Subsystems* that are known to *Sm*.

SmApiPort - An output variable that returns a handle to a communication port connected to *Sm*, and over which the **Sm...** APIs may be made.

The **SmConnectToSm** API is provided so that *Emulation Subsystem's* and *Logon Processes* can connect to *Sm*. For *Emulation Subsystem's*, the *SbApiPortName,* and *SbApiPort* parameters must be supplied. This is because in addition to creating a connection to *Sm* (over which the **Sm...** APIs are exported), a connection is made to the *Emulation Subsystem* over which the **Sb...** APIs are exported.

2.2 SmGetLogonObjectDirectory

NTSTATUS
SmGetLogonObjectDirectory(
    IN ULONG *SessionId* **OPTIONAL**,
    OUT PSTRING *LogonObjectDirectoryName*
    );

Parameters:

SessionId - An optioanl variable that supplies the session id whose associated logon object directory name is to be found. If this optional parameter is not provided, then the caller's logon object directory name is returned.

LogonObjectDirectoryName - A variable that returns the name of the session's associated logon object directory.

The name of the logon object directory associated with a session can be determined using the **SmGetLogonObjectDirectory** function.

3. Logon Process Support

Before a user can make use of the **Windows NT** system, that user must first "logon" to the system. Device-specific logon processes are responsible for collecting information about the user and authenticating the user. The authentication is performed using services of the Local Security Authority. Following authentication, a logon process may decide to activate a user interface shell program to interact with the user.[1] This is done using *Sm* services.

The *Sm* services provided to support logon processes are:

SmRegisterLogonProcess
SmExecLogonShell

[1] Note that this is not always the case. The LAN Manager logon process, for instance, authenticates users as part of session setup, but no shell process is activated.

These services are described in following subsections. Before these API descriptions, some background/philosophy informtion is provided on logon processes.

3.1 Logon Process Philosophy

The general philosophy and logic of logon processes, from the perspective of the *Sm* is:

- Some set of logon process are activated by configuration control or other means. For the standard Windows NT devices (windows, terminals, LAN Manager), the logon processes will be started as part of device/network initialization. Other logon processes, such as automated teller device, or cash-register device logon processes may be started either via configuration control, or other mechanisms, such as operator actions.

  Note that there is nothing special about a logon process except that it has the **SeTcbPrivilege** privilege. Note also that a logon process does not have to be an independent process running nothing but logon process code. For example, the windows server (User32 server) could include logon processing code within it.

- Each logon process connects to the session manager using **SmConnectToSm()**. The *SbApiPortName* is left null in this call to indicate that something other than an emulation subsystem is connecting. At this time, the session manager doesn't yet know that the connected client is a logon process.

- The logon process then identifies itself as a logon process. This is done using the **SmRegisterLogonProcess()** API. This allows the session manager to authenticate the caller as having the **SeTcbPrivilege**.

  As part of **SmRegisterLogonProcess()** processing, the session manager opens the client process for **PROCESS_DUP_HANDLE** access. Note that all calls from this logon process must originate from this same process. That is, the port object handles used to communicate with the session manager can not be shared with a third process who will also act as a logon process.

- When a user attempts to log on, the logon process collects identification and authentication information and calls the Local Security Authority (LSA) directly to authenticate the user. If the authentication is successful, the logon process will be given a handle to a primary token representing the new logon session.
- Once a user has been successfully authenticated, the logon process may activate a root process for the user by calling **SmExecLogonShell()**. This call takes as parameters:
    - The name of the shell (image) to activate,
    - A handle to the primary token to assign to the new process,
    - Memory quota information for the new process,
    - A GUID representing the new logon session (which the session manager will use to create a logon object directory),
    - (optional) environment variables that are to be passed to the new logon shell process.
- The session manager attempts to create a new process running the logon shell image. The session manager sets the process's primary token to be that supplied by the logon process. The initial thread of this process is created, but left in a suspended state. It is the logon process's responsibility to resume the thread when desired.

  If the process creation is successful, then handles to the newly created shell process and thread are returned to the logon process. The process handle will be open for **SYNCHRONIZE** access. The thread handle will be open for **THREAD_SUSPEND_RESUME** access. Logon processes are expected to close these handles when no longer needed.

This allows logon processes to:

1) Specify UI shell initialization parameters (via environment variables). For example, the User32 logon process will specify the name of the

window station the user has logged on from using environment variables.

2) Wait on the newly logged on process to exit unexpectedly. For example, a windows32 logon shell is expected to open a desktop object in the window station the user logged on from. If the shell process exits before openning a desktop, then the User32 logon process assumes something has gone wrong and treats the condition as a logoff, making the window station available for another logon.

3.2 SmRegisterLogonProcess

NTSTATUS
SmRegisterLogonProcess(
    IN HANDLE *SmApiPort,*
    IN PSTRING *LogonProcessName*
    );

Parameters:

SmApiPort - A variable that supplies an handle to a communcation port connected to *Sm*.

LogonProcessName - A name string that identifies the logon process. This should be a printable name suitable for display to administrators. For example, "User32LogonProcess" might be used for the windows logon process name. No check is made to determine whether the name is already in use.

Return Value:

STATUS_SUCCESS - The call completed successfully.

STATUS_PRIVILEGE_NOT_HELD - Indicates the caller does not have the privilege necessary to act as a logon process. **SeTcbPrivilege** is needed.

Before being able to use the **SmExecLogonShell()** service, a logon process must identify itself as a logon process. This is done using the **SmRegisterLogonProcess()** service.

This service verifies that the caller is a legitimate logon process. This is done by ensuring the caller has **SeTcbPrivilege**. It also opens the caller's process for **PROCESS_DUP_HANDLE**. This information is cached for future use.

3.3 SmExecLogonShell

NTSTATUS
SmExecLogonShell(
IN HANDLE *SmApiPort,*
IN GUID *LogonGuid*,
IN PSTRING *ShellImageName*,
IN HANDLE *PrimaryToken*,
IN QUOTA_LIMITS *Quotas*,
IN RTL_USER_PROCESS_PARAMETERS *ProcessParameters*,
OUT PHANDLE *Process*,
OUT PHANDLE *Thread*
);

Parameters:[2]

SmApiPort - A variable that supplies an handle to a communcation port connected to *Sm*.

LogonGuid - A GUID uniquelly assigned to represent this logon session.

ShellImageName - Provides the path name of the shell program to execute.

PrimaryToken - Provides a handle to the primary token to assign to the new process. This handle must be open for **TOKEN_ASSIGN_PRIMARY** access.

Quotas - Provides quota values to be assigned to the new process.

ProcessParameters - Provides parameters to be passed to the new process.

Process - Receives a handle to the new process. The handle will be open for **SYNCHRONIZE** access.

[2] Loup, DaveC, DarrylH: Do we need a *CaptiveAccount* parameter too?

Thread - Receives a handle to the initial thread of the process. The handle will be open for **THREAD_SUSPEND_RESUME** access. The thread will not yet have been activated.

Return Value:

STATUS_SUCCESS - The call completed successfully.

STATUS_NOT_LOGON_PROCESS - The caller has not registered as a logon process.

STATUS_LOGON_SESSION_EXISTS - Indicates the GUID assigned to this logon session is already in use.

In addition to these, the following general classes of errors may be returned:

- Errors related to creation of a process or thread, including attempts to access the image file.
- Attempts to duplicate and assign the primary token.

This service is used by logon processes to activate a user interface shell program for a newly logged on interactive user. The logon process may pass information to the new shell program via environment variables.

The session manager:

1) Creates a new logon session to run the logon shell program in,
2) Creates a logon object directory for the new logon session,
3) creates the logon program and the initial thread in that program (but leaves the thread in a suspended state).

Handles to the new process and its initial thread are passed back to the requesting logon process. The process handle will be open for **SYNCHRONIZE** access. The thread handle will be open for **THREAD_SUSPEND_RESUME** access. The logon process is expected to close these handles when no longer needed.

4. System Subsystems Support

System subsystems are logical extensions of the operating system. They provide privileged and protected operating system support, but are implemented as separated processes that execute in user mode.

4.1 Session Control Services

The *Sm* subsystem is responsible for coordinating the creation and management of sessions. It is responsible for coordinating the creation of sessions when *Emulation Subsystems* encounter an image file designed to operate in a different API environment.

Sm tends to act as an intermediary between *Emulation Subsystems*. It is responsible for allocating session ID's, and for associating a session ID with its controlling *Emulation Subsystem*.

Sm is also responsible for associating an image file with the *Emulation Subsystem* it is designed to run with.

Sm exports the following APIs to support *Emulation subsystem* operations:

SmCreateForeignSession
SmSessionComplete
SmTerminateForeignSession

4.1.1 SmCreateForeignSession

A request to create a foreign session can be made using the **SmCreateForeignSession** function.

NTSTATUS
SmCreateForeignSession(
IN HANDLE *SmApiPort,*
OUT PULONG *ForeignSessionId,*
IN ULONG *SourceSessionId,*
IN PRTL_USER_PROCESS_INFORMATION *ProcessInformation,*
IN PCID *DebugUiClientId* **OPTIONAL**,
IN HANDLE *StandardInput* **OPTIONAL**,
IN HANDLE *StandardOutput* **OPTIONAL**,
IN HANDLE *StandardError* **OPTIONAL**
);

Parameters:

SmApiPort - A variable that supplies an handle to a communcation port connected to *Sm*.

ForeignSessionId - A variable whose return value specifies the session ID of the created session. The session ID is assigned by the session manager. The session ID is used in the session control APIs to identify the target foreign session.

SourceSessionId - A variable that specifies the session ID of the application that is creating (through its *Emulation Subsystem*) the foreign session. This session ID is used by *Sm* to determine a user profile for the new session.

ProcessInformation - A structure that describes the process to be run as a foreign session. This data structure contains a complete description of the process including handles to the process and its initial thread. Using **NtDupObject**, *Sm* makes these handles

available to the *Emulation Subsystem* responsible for the process. Regardless of the outcome of this call, the calling process looses its handles to the process and thread.

DebugUiClientId - An optional parameter that specifies the client ID of the debugger user interface that is debugging the session. If this parameter is specified, then the session is a "debug session".

StandardInput - An optional handle that specifies the standard input stream associated with the session. Using **NtDupObject**, *Sm* makes this handle available to the *Emulation Subsystem* responsible for the process. Regardless of the outcome of this call, the calling process' version of this handle is closed.

StandardOutput - An optional handle that specifies the standard output stream associated with the session. Using **NtDupObject**, *Sm* makes this handle available to the *Emulation Subsystem* responsible for the process. Regardless of the outcome of this call, the calling process' version of this handle is closed.

StandardError - An optional handle that specifies the standard error output stream associated with the session. Using **NtDupObject**, *Sm* makes this handle available to the *Emulation Subsystem* responsible for the process. Regardless of the outcome of this call, the calling process' version of this handle is closed.

Emulation Subsystems use this service whenever they are instructed to execute an image whose type is not supported by the subsystem (e.g. an **OS/2** application executes a **DosExecPgm** specifying an image file that is a **POSIX** application).

Sm implements this API by associating the image file type with an appropriate *Emulation Subsystem*, allocating a new session ID, transfering the handles (*Thread, Process, StandardInput, StandardOutput,* and *StandardError*) into the appropriate *Emulation Subsystem's* handle table, and calling the *Emulation Subsystem* at its **SbCreateSession** entry point. Assuming that the call to **SbCreateSession** succeeds, the session ID of the new session is returned to the caller.

4.1.2 SmSessionComplete

Sm is notified that a session has completed through the **SmSessionComplete** function.

NTSTATUS
SmSessionComplete(
    IN HANDLE *SmApiPort,*
    IN ULONG *SessionId,*
    IN NTSTATUS *CompletionStatus*
    );

Parameters:

SmApiPort - A variable that supplies an handle to a communcation port connected to *Sm*.

SessionId - A parameter that specifies the session ID of the foreign session that has completed.

CompletionStatus - A parameter that specifies the completion status of the session.

The **SmSessionComplete** API is provided so that an *Emulation Subsystem* can notify *Sm* that one of its sessions has completed.

Once *Sm* receives this call, it locates the *Emulation Subsystem* that created the foreign session using the specified session ID, and calls the subsystem at its **SbForeignSessionComplete** entry point.

4.1.3 SmTerminateForeignSession

A request that a foreign session be terminated can be made through the **SmTerminateForeignSession** function.

NTSTATUS
SmTerminateForeignSession(
    IN HANDLE *SmApiPort,*
    IN ULONG *ForeignSessionId,*
    IN NTSTATUS *TerminationStatus*
    );

Parameters:

SmApiPort - A variable that supplies an handle to a communcation port connected to *Sm*.

ForeignSessionId - A parameter that specifies the session ID of the foreign session being terminated.

TerminationStatus - A parameter that specifies the reason that the foreign session should be terminated.

The **SmTerminateForeignSession** API is provided so that an *Emulation Subsystem* can request the termination of a foreign session that it created.

Sm implements this call by locating the appropriate *Emulation Subsystem* using the specified foreign session ID, and then calling the subsystem at its **SbTerminateSession** entry point.

The **SmTerminateForeignSession** call returns before the session is actually terminated. When the session terminates *Sm* will be notified.

4.2 Piper

The *Piper* subsystem is responsible for providing pipe stream input and output between threads in different sessions (under the supervision of *Emulation Subsystems*).

This capability is provided to support transfering information between applications that are of a different class (e.g *foo* | *bar* where *foo* is a **POSIX** application and *bar* is and **OS/2** application).

Piper requires coordination between the *Emulation Subsystem* involved in the data piping, and the application runtime libraries that provide stream input and output through the *STDIN, STDOUT,* and *STDERR* I/O streams. All application input and output through these streams must be handled by the application's *Emulation Subsystem*. Only the subsystem knows the session that the application is part of, and the "file names" of its input, output, and error streams.

Piper exports the following APIs:

PiperCreatePipe
PiperJoinPipe
PiperLeavePipe
PiperReadPipe
PiperWritePipe

4.2.1 PiperCreatePipe

An *Emulation Subsystem* creates the potential for pipe stream communication between the application threads in one of its sessions, and application threads in a "foreign" session that it asked *Sm* to create using the **PiperCreatePipe** function.

NTSTATUS

PiperCreatePipe(

IN ULONG *ForeignSessionId*,
IN ULONG *SourceSessionId*
);

Parameters:

ForeignSessionId - Specifies the session ID of the foreign session that makes up the other end of the pipe.

SourceSessionId - Specifies the session ID of the local session that is creating the pipe.

Creating a pipe causes the potential for pipe stream communication to occur between the two specified sessions. Pipes provide a full duplex byte stream communication path between application threads in the specified sessions.

Data written by application threads within the local session is made available (to satisfy pipe reads) to threads within the foreign session. Reads to the pipe by application threads within the local session are satisfied by corresponding pipe writes made by threads within the foreign session.

After this call completes, application threads within the local session may attempt to read data from the pipe, and write data to the pipe. Until the foreign session joins the pipe using the **PiperJoinPipe** API, data that they write will remain in the pipe, and their pipe reads will block.

There is no need to synchronize this call with a corresponding **PiperJoinPipe** call specifying the foreign session. These calls may be issued in either order.

4.2.2 PiperJoinPipe

An *Emulation Subsystem* joins a pipe so that it can participate in pipe stream communication between the application threads in one of its sessions, and application threads in the session that created it using the **PiperJoinPipe** function.

NTSTATUS
PiperJoinPipe(

IN ULONG *SessionId*
);

Parameters:

SessionId - Specifies the session ID of the local session that is joining a pipe.

Joining a pipe allows the threads within the specified local session to begin pipe stream communication over a pipe created in a corresponding call to **PiperCreatePipe**.

After this call completes, application threads within the local session may read data from the pipe, and write data to the pipe.

This call completes when a corresponding call to **PiperCreatePipe** is issued specifying the local session in its *ForeignSessionId* parameter.

4.2.3 PiperLeavePipe

An *Emulation Subsystem* leaves a pipe which informs *Piper* that it no longer wants to participate in pipe stream communication using the **PiperLeavePipe** function.

NTSTATUS

PiperLeavePipe(

IN ULONG *SessionId*
);

Parameters:

SessionId - Specifies the session ID of the local session that is leaving the pipe.

Leaving a pipe causes application threads within the local session to disassociate themselves with the pipe. All data destined for the local session is flushed, and further pipe writes to the local session fail.

When both sessions that make up a pipe leave the pipe, the pipe is deleted. All data within or destined for the pipe is deleted.

4.2.4 PiperReadPipe

An *Emulation Subsystem* can read data from a pipe stream that it has either joined or created using the **PiperReadPipe** function.

NTSTATUS

PiperReadPipe(

IN ULONG *SessionId*,
OUT PUCHAR *DataReadBuffer*,
IN ULONG *DataReadLength*,
OUT PULONG *DataActuallyReadLength*
);

4.2.5 PiperWritePipe

An *Emulation Subsystem* can write data to a pipe stream that it has either joined or created using the **PiperWritePipe** function.

NTSTATUS

PiperWritePipe(

IN ULONG *SessionId*,
IN PUCHAR *DataWriteBuffer*,
IN ULONG *DataWriteLength*,
OUT PULONG *DataActuallyWrittenLength*
);

5. Emulation Subsystems

The primary role of an *Emulation Subsystem* is to emulate a set of system services using native **Windows NT** system services. Applications written to a particular API use the appropriate *Emulation Subsystem* to implement that particular system API.

Each application contains in its image file header, a description of the *Emulation Subsystem* that the application requires (e.g. OS/2 applications like cmd.exe describe the **OS/2++** subsystem). In addition to providing operating system API emulation, the subsystem is responsible for managing the session to which the application belongs. The subsystem also acts as an intermediary between the **Dbg** protected subsystem and the application when the application is being "debugged".

Each *Emulation Subsystem* exports three **Windows NT** connection ports. An *LPC* connection to an *Emulation Subsystem* is established by specifying one of these ports in a call to **NtConnectPort**. Each connection port is associated with a class of services implemented by the *Emulation Subsystem*. The three classes of services are:

- *Sm* to *Emulation Subsystem* APIs. The connection port associated with this class of services is protected such that only the *Sm* subsystem can access the port. Once a connection has been established, the *Emulation Subsystem* does not respond to connection requests arriving on this port. The connection between *Sm* and each *Emulation Subsystem* is a trusted connection.

- *Dbg* to *Emulation Subsystem* APIs. The connection port associated with this class of services is protected such that only the *Dbg* subsystem can access the port. Once a connection has been established, the *Emulation Subsystem* does not respond to connection requests arriving on this port. The connection between *Dbg* and each *Emulation Subsystem* is a trusted connection.

- Operating System APIs emulated by the subsystem. The connection port associated with this class of services does not have to be protected. Each application that is using the APIs exported over this connection establishes a connection during its process initialization sequence (*crt0* equivalent). *Emulation Subsystem* must

authenticate each call (associate the caller's CID with a CID created by the subsystem) to ensure that the thread making the call is one of its threads. The connection between and application and its *Emulation Subsystem* is not a trusted connection.

5.1 PSX++

The **PSX++** protected subsystem implements the APIs described in the *IEEE P1003.1/Draft 13 August 22, 1988* specification. It is responsible for managing all applications written to this API.

5.2 OS/2++

The **OS/2++** protected subsystem implements the *Cruiser OS/2 V2.0* APIs. It is responsible for managing all applications written to this API.

5.3 NT++

The **NT++** protected subsystem implements a very small set of APIs. Its primary purpose is to implement the set of APIs needed to manage and control sessions, and to provide a *DosExecPgm* like API that a native debugger user interface or application can use to create and manage a session or to execute an image designed to run with one of the other *Emulation Subsystems*.

5.4 Emulation Subsystem APIs

Each *Emulation Subsystem* exports a set of APIs designed to manage and control sessions. These APIs are called by the *Sm*, *Dbg*, or by the **Windows NT** executive.

Emulation Subsystems export the following APIs:

SbCreateSession
SbTerminateSession
SbForeignSessionComplete

Emulation Subsystems see the APIs in their raw form. The subsystems must provide their own "API Loops" that receive and reply using *LPC* messages. The subsystem APIs are all called (by their own API loops) with a pointer to a subsystem API message (*SBAPIMSG*) the format of the message is given below:

SbApiMsg Structure

PORTMSG *h* - This field contains a standard *LPC* port message. The *ClientId* of the sender, message type, and length information are all placed in this area by the system.

SBAPINUMBER *ApiNumber* - This field specifies the API number of the call. Values are:

ApiNumber Enumeration

SbCreateSessionApi - The message specifies the **SbCreateSession** API.

SbTerminateSessionApi - The message specifies the **SbTerminateSession** API.

SbForeignSessionCompleteApi - The message specifies the **SbForeignSessionComplete** API.

NTSTATUS *ReturnedStatus* - This field is used to pass the return status of the **Sb...** API back to the caller of the API. This field is designed to be modified by the "API loop".

union *u* - This union contains one field for each of the API types.

u Union

SBCREATESESSION *CreateSession* - This field contains information specific to the **SbCreateSession** API.

SBTERMINATESESSION *TerminateSession* - This field contains information specific to the **SbTerminateSession** API.

SBFOREIGNSESSIONCOMPLETE *ForeignSessionComplete* - This field contains information specific to the **SbForeignSessionComplete** API.

5.4.1 SbCreateSession

A session is created and placed under the control of an *Emulation Subsystem* through the **SbCreateSession** function.

NTSTATUS
SbCreateSession(
    IN OUT PSBAPIMSG *SbApiMsg*
    );

Parameters:

SbApiMsg - A variable that supplies an *LPC* message that contains information necessary to allow the subsystem to create a session capable of running the process described in the message.

The *ApiNumber* associated with this call is *SbCreateSessionApi*. The *CreateSession* field of the API message contains the following:

CreateSession Structure

ULONG *SessionId* - A variable that specifies the session ID to be associated with the session being created. The session ID is assigned by the session manager. The session ID is used in the session control APIs to identify the target session.

RTL_USER_PROCESS_INFORMATION *ProcessInformation* - A structure that describes the process to be run as a new session. This data structure contains a complete description of the process including handles to the process and its initial thread. The subsystem is responsible for the process and thread even if it fails the create session request. It must terminate and close the process and thread at the appropriate time (even if it fails the session creation).

CID *DebugUiClientId* - An optional parameter that specifies the client ID of the debugger user interface that is debugging the session. If this parameter is specified, then the session is created as a

"debug session". Debug sessions are created in a suspended state (i.e., the initial thread of the process is left suspended). In addition, the subsystem servicing this call must call into the *Dbg* subsystem to report the new debug session and the CID of the debugger user interface that is debugging the session.

// All Windows NT threads are created in a suspended state. Most Emulation Subsystems create an application thread by creating a Windows NT thread and then resuming the thread. This parameter instructs the Emulation Subsystem to not resume the application thread. The Emulation Subsystem will be instructed to resume the thread through a DebugUi -> Dbg -> Emulation Subsystem transaction. //

The value of this parameter originates in the system. When a *DebugUi* issues a call to an API that creates a "debug process" the CID of the *DebugUi* is captured by the *DebugUi's Emulation Subsystem* from the message header of the message associated with the process creation call. If the process is foreign to the *DebugUi's* subsystem, the CID passes from the *DebugUi's Emulation Subsystem* to *Sm*, and then from *Sm* to the *Emulation Subsystem* that should run the process.

HANDLE *StandardInput* - An optional handle that specifies the standard input stream associated with the session. Regardless of the outcome of this call, the subsystem is responsible for closing the handle at the appropriate time (even if it fails the session creation).

HANDLE *StandardOutput* - An optional handle that specifies the standard output stream associated with the session. Regardless of the outcome of this call, the subsystem is responsible for closing the handle at the appropriate time (even if it fails the session creation).

HANDLE *StandardError* - An optional handle that specifies the standard error output stream associated with the session. Regardless of the outcome of this call, the subsystem is responsible for closing the handle at the appropriate time (even if it fails the session creation).

The *Sm* subsystem uses the **SbCreateSession** API to create a session to run the specified process. This call is made as part of the logon sequence (part of **SmLogonUser**), or when *Sm* is asked (by another subsystem, termed the "source subsystem") to create a session to run an image whose format is not understood by the source subsystem.

5.4.2 SbTerminateSession

A session can be terminated through the **SbTerminateSession** function.

NTSTATUS
SbTerminateSession(
    IN OUT PSBAPIMSG *SbApiMsg*
    );

Parameters:

SbApiMsg - A variable that supplies an *LPC* message that contains information necessary to allow the subsystem to terminate the specified session.

The *ApiNumber* associated with this call is *SbTerminateSessionApi*. The *TerminateSession* field of the API message contains the following:

TerminateSession Structure

ULONG *SessionId* - A value that specifies the session ID of the session being terminated.

NTSTATUS *TerminationStatus* - A that specifies the reason that the session should be terminated.

The **SbTerminateSession** API is provided so that a session can be terminated. This call is made by the *Sm* subsystem in response to a request by the *Emulation Subsystem* that indirectly created the session.

The **SbTerminateSession** call returns before the session is actually terminated. When the session terminates, the *Sm* subsystem will be notified through an RPC from the session's controlling *Emulation Subsystem* to *Sm* at its **SmSessionComplete** entry point.

5.4.3 SbForeignSessionComplete

An *Emulation Subsystem* is notified that a foreign session has completed through the **SbForeignSessionComplete** function.

NTSTATUS
SbForeignSessionComplete(
    IN OUT PSBAPIMSG *SbApiMsg*
    );

Parameters:

SbApiMsg - A variable that supplies an *LPC* message that contains information which notifies the subsystem that a foreign session that it started has completed.

The *ApiNumber* associated with this call is *SbForeignSessionCompleteApi*. The *ForeignSessionComplete* field of the API message contains the following:

ForeignSessionComplete Structure

ULONG *SessionId* - A value that specifies the session ID of the session that has completed.

NTSTATUS *CompletionStatus* - A value that specifies the completion status of the session.

The **SbForeignSessionComplete** API is provided so that a subsystem can be notified that a foreign session that it created has completed. The subsystem that services this call is the subsystem that originally requested that the foreign session be created.

Once this call returns, the session ID is available for re-use.

Revision History

Revision 1.9, January 7, 1990. Jim Kelly (JimK)

1) Eliminated all references to Presentation Manager.

2) Changed logon so that logon processes authenticate directly with the Local Security authority (LSA) and then interact with the NT Session Manager to activate the logon shell process. This obsoleted the SmLogonUser() API and caused the introduction of the SmRegisterLogonProcess() and SmExecLogonShell() APIs.