Portable Systems Group

NT OS/2 System Startup Design Note

Author: *Mark Lucovsky*

Revision 1.2, July 26, 1990
Original Draft May 31, 1990

1. Overview

This design note describes system startup for NT OS/2. For the purposes of this paper, system startup begins after phase 1 system initialization completes.

This paper does not describe an agreed to, long term strategy. It describes an interim startup sequence that will be used until the system installation and system configuration plans solidify.

1.1 System Initialization

This paper does not describe system initialization in any great level of detail. System initialization occurs in three phases. After boot loading the system image, the kernel is called at **KiSystemStartup**. After kernel initialization, the executive is called to perform phase 0 initialization. This causes memory management, the object manager, and the process architecture to initialize. During phase 0, a thread is created to perform phase 1 initialization. Upon completion of phase 0 initialization, a context switch to the thread occurs and it proceeds with phase 1 initialization. The bulk of NT OS/2 is initialized during this phase. Upon completion of phase 1 initialization, system startup begins.

1.2 System Startup

System startup occurs in the NT OS/2 executive, and in the NT OS/2 Session Manager (SM).

1.2.1 Executive Level System System Startup

The executive's role in system startup is very simple. It simple creates a process to run SM. This is done using **RtlCreateUserProcess** with an image name of "\BootDevice\smss.exe". If the executive fails to create a process to run SM, then system startup fails. If this occurs on debug systems (compiled with DBG set), the NT OS/2 Command Line Interpreter (CLI) is started; otherwise, the system is halted with a bug check.

1.2.2 Session Manager System Startup

If executive level system startup is successful, then SM starts. SM begins by initializing itself. This includes:

- Creating ports for session manager, and subsystem APIs.
- Creating listen and API threads for session manager, and subsystem APIs.
- Enabling all connection requests and APIs.

Once SM is initialized, it begins system startup. Session Manager level system startup is driven from the NT OS/2 system configuration file located in "\BootDevice\ntos2.cfg". The file is a standard configuration file operated on by the "RtlxxxxConfigFile" APIs (described later in this paper).

Session Manager level system startup occurs in two phases. During the first phase, the "soft", or "configurable" portion of the NT OS/2 executive is configured. During the second phase, the session manager starts and initializes various protected subsystems.

1.2.2.1 First Phase SM system startup

First phase SM system startup begins with an open of the NT OS/2 system configuration file. This is done using **RtlOpenConfigFile** with a pathname of "\BootDevice\ntos2.cfg". If the open fails, then SM system startup does not occur. The system will run, but system level subsystems, and other portions of the system normally controlled by SM startup will not occur.

Once the configuration file is opened, the "ntos2" section is located using **RtlLocateSectionConfigFile** with a section name of "ntos2". If this section can not be located, then first phase SM system startup terminates and second phase SM system startup begins.

If the "ntos2" section is located, then all of the keywords in the section are enumerated and processed. Keywords are processed in configuration file order. Multiple keywords (keywords with the same keyword name) are processed in configuration file order.

SM reckognizes a small set of "ntos2" section keywords. The following sections describe the set of supported keywords.

1.2.2.1.1 Link Keyword

The "Link" keyword causes SM to create a symbolic link object accessible to all processes in the system. Link keywords are processed in configuration file order. The syntax of the link keyword is as follows:

```
Link = NAME-OF-LINK NAME-OF-LINK-TARGET
```

The "standard" set of symbolic link objects can be created by including the following in the "ntos2" section of your configuration file. Note that system initialization no longer creates these symbolic link objects. It only creates the "\BootDevice" symbolic link.

```
[ntos2]
//
// MIPS Links
//
// Link = \A: \Device\SaDisk
// Link = \C: \Device\HardDisk   // for 3240
// Link = \C: \Device\SaDisk     // for sable
//
//
// Simulator Links
//
// Link = \A: \Device\Floppy1
// Link = \B: \Device\Floppy2
// Link = \C: \Device\Filesystem
// Link = \D: \Device\Filesystem    // ddfs
// Link = \C: \Device\HardDisk1     // pinball
```

```
//
//
// Standard Links
//
Link = \A: \Device\Floppy0
Link = \C: \Device\HardDisk0\Partition1
Link = \D: \Device\HardDisk1\Partition1
Link = \E: \Device\HardDisk2
Link = \SystemDisk \C:
```

1.2.2.1.2 PagingFile Keyword

The "PagingFile" keyword causes SM to create a system wide file used by the modified page writer. If a paging file keyword does not exist in your system configuration file, then the amount of virtual memory available on the system is limited. System initialization no longer creates a paging file. The syntax of the pagingfile keyword is as follows:

```
PagingFile = NAME-OF-PAGING-FILE MAXIMUM-SIZE-OF-PAGING-FILE-MEGABYTES
```

The following example causes the system to use a 10Mb paging file called pagefile.sys in the "\Nt" directory of the system disk.

```
[ntos2]
//
// Create a 10Mb Paging File
//
PagingFile = \SystemDisk\Nt\pagefile.sys 10
```

Note that since the previous example uses the "\SystemDisk" symbolic link, it must occur after the appropriate link keyword.

1.2.2.2 Second Phase SM system startup

Second phase SM system startup begins after first phase SM startup completes. If first phase SM fails to open the NT OS/2 system configuration file, then this phase is skipped.

This phase begins by locating the "sm" section of the configuration file. This is done using **RtlLocateSectionConfigFile** with a section name of "sm". If this section can not be located, then second phase SM system startup terminates.

If the "sm" section is located, then all of the keywords in the section are enumerated and processed. Keywords are processed in configuration file order. Multiple keywords (keywords with the same keyword name) are processed in configuration file order.

SM reckognizes a small set of "sm" section keywords. The following sections describe the set of supported keywords.

1.2.2.2.1 Subsystem Keyword

The "SubSystem" keyword causes SM to start the specified system service emulation subsystem as a registerd subsystem. The subsystem must conform to the subsystem/SM connection protocol, and must accept appropriate proceses.

When SM encounters a subsystem keyword, it creates a process to run the subsystem, and waits for the subsystem to complete the connection protocol.

The syntax of the subsystem keyword is as follows:

```
SubSystem = [debug ] PATHNAME-OF-SUBSYSTEM [, OPTIONAL-COMMAND-LINE]
```

Note that processes created through the SubSystem keyword must be marked as COFF_TARGET_SUBSYSTEM_NATIVE images.

The debug prefix is optional and if present invokes the subsystem with the DebugFlag passed to main set to 1. If the optional command line test is specified, then a pointer to the text after the comma will be passed in argv[1] to the main procedure of the subsystem.

The following example shows the subsystem keyword needed to start the OS/2 subsystem.

```
[sm]
//
// Start the os2 subsystem
//
SubSystem = \SystemDisk\Nt\SubSys\os2ss.exe
```

1.2.2.2.2 Start Keyword

The "Start" keyword causes SM to create and start a process. SM then waits for the initial thread in the process to terminate. Using this mechanism, you can write a server program whose initial thread performs all initialization then creates its worker threads. When the server is fully initialized and all worker threads are ready, the initial thread in the process could terminate. This allows SM to begin processing other keywords.

The syntax of the start keyword is as follows:

```
Start = [debug ] PATHNAME-OF-PROGRAM-TO-START [, OPTIONAL-COMMAND-LINE]
```

Note that processes created through the start keyword must be marked as COFF_TARGET_SUBSYSTEM_NATIVE images.

The debug prefix is optional and if present invokes the process with the DebugFlag passed to main set to 1. If the optional command line test is specified, then a pointer to the text after the comma will be passed in argv[1] to the main procedure of the process.

The following example causes SM to start the Presentation Manager (PM) and security servers.

```
[sm]
//
// Start PM
//
Start = \SystemDisk\Nt\SubSys\pmsrv.exe
//
// Start SAM
//
start = \SystemDisk\Nt\SubSys\samsrv.exe
```

1.2.2.2.3 Run Keyword

The "Run" keyword causes SM to create and start a process. SM does not wait for the process. It is simply created and set free to run. This keyword is useful to start logon processes and shells, and various "leaf" system processes.

The syntax of the run keyword is as follows:

```
Run = [debug ] PATHNAME-OF-PROGRAM-TO-RUN [, OPTIONAL-COMMAND-LINE]
```

Note that processes created through the run keyword must be marked as COFF_TARGET_SUBSYSTEM_NATIVE images.

The debug prefix is optional and if present invokes the process with the DebugFlag passed to main set to 1. If the optional command line test is specified, then a pointer to the text after the comma will be passed in argv[1] to the main procedure of the process.

The following example causes SM to run the PM Shell and NT OS/2 Error Logger.

```
[sm]
//
// Start PM Shell
//
Run = \SystemDisk\Nt\Bin\pmshell.exe
//
// Start Error Logger
//
Run = \SystemDisk\Nt\Bin\errlog.exe
```

1.2.2.2.4 Debug Keyword

The "Debug" keyword causes SM to start the debug subsystem and enable debugging.

The syntax of the debug keyword is as follows:

```
Debug = PATHNAME-OF-DEBUG-SUBSYSTEM
```

The following example causes SM to start the debug subsystem.

```
[sm]
//
// Start DBG
//
```

```
Debug = \SystemDisk\Nt\SubSys\dbgss.exe
```

1.2.2.2.5 LibPath Keyword

The "LibPath" keyword causes SM to establish a default DLL search path. This path is made available to the loader subsystem and is used to locate DLLs.

The syntax of the LibPath keyword is as follows:

```
LibPath = DLL-SEARCH-PATH
```

The following example illustrates the use of the LibPath keyword:

```
[sm]
//
// Establish a DLL search path
//
LibPath = \BootDevice;\SystemDisk\Nt\Dll
```

1.2.2.2.6 Short Term Keywords

There are a number of keywords that are currently supported by SM, but are only supported due to shortcomings in the rest of the system. These keywords will remain until the appropriate components are complete and they are no longer needed. The following sections describe keywords that are supported, but have a limited lifetime.

1.2.2.2.6.1 GlobalFlag Keyword

The "GlobalFlag" keyword causes SM to set the value of *NtGlobalFlag*. This flag controls various debug portions of the system.

The syntax of the GlobalFlag keyword is as follows:

```
GlobalFlag = VALUE-FOR-NTGLOBALFLAG
```

The following example illustrates the use of the GlobalFlag keyword:

```
[sm]
//
// Setup NtGlobalFlag to display object deletion messages
// and to stop on first chance exceptions
//
GlobalFlag = 3
```

1.2.2.2.6.2 Path Keyword

The "Path" keyword causes SM to establish a default search path used when running programs from the NT SM> CLI.

The syntax of the Path keyword is as follows:

```
Path = DEFAULT-SEARCH-PATH
```

The following example illustrates the use of the Path keyword:

```
[sm]
//
// Setup a search path used at the NT SM> prompt
//
Path = \BootDevice;\SystemDisk\Nt\Bin
```

1.2.2.2.6.3 Quota Keywords

The quota keywords establish a default quota for processes started from the NT SM> CLI. Quota keywords can be used to establish:

- Non-paged pool limit
- Paged pool limit
- Minimum working set size
- Maximum working set size
- Pagefile limit

If a configuration file does not specify a quota keyword, then the defult for that resource is unlimited. Quota keywords have the following syntax:

```
PagedPoolLimit = AMOUNT-OF-PAGED-POOL-QUOTA
NonPagedPoolLimit = AMOUNT-OF-PAGED-POOL-QUOTA
MinimumWorkingSetSize = MINIMUM-WORKING-SET-SIZE-IN-PAGES
MaximumWorkingSetSize = MAXIMUM-WORKING-SET-SIZE-IN-PAGES
PagefileLimit = MAXIMUM-PAGE-FILE-USAGE
```

The following example illustrates the use of the quota keywords:

```
[sm]
//
// Setup Quota
//
PagedPoolLimit = 1048576
NonPagedPoolLimit = 1048576
MinimumWorkingSetSize = 45
MaximumWorkingSetSize = 75
```

2. Configuration File APIs

The NT OS/2 user-mode DLL (udll.dll) contains a set of APIs used to manage configuration files. Configuration files are very similar to existing Microsoft configuration files (tools.ini, win.ini...).

Configuration files consist of sections. Within sections, there are keywords that have optional values. The following describes the format of a configuration file (note that "#", or "//" begin line comments).

```
//
// Section names are enclosed in "[" and "]". Section names are
// any valid C identifier. Section names are case insensitive.
//
[NAME-OF-SECTION]
// Keywords appear within sections.  Keyword names are any
// valid C identifier and are case insensitive.  Keywords
// may optionally contain a value.  A keyword's value is all
// characters to the right of the "=" character.  Leading
// and trailing whitespace is trimmed.  If a line comment
// occurs on a keyword line, the keyword's value ends at
// the start of the line comment with trailing whitspace
// trimmed.
NAME-OF-KEYWORD                 // keyword without a value
NAME-OF-KEYWORD = VALUE-OF-KEYWORD  // keyword with a value
```

The following sample shows a standard NT OS/2 configuration file.

```
[ntos2]
//
// Standard Links
//
Link = \A: \Device\Floppy0
Link = \C: \Device\HardDisk0\Partition1
Link = \D: \Device\HardDisk1\Partition1
Link = \E: \Device\HardDisk2
Link = \SystemDisk \C:
//
// Create a 10Mb Paging File
//
PagingFile = \SystemDisk\Nt\pagefile.sys 10
[sm]
//
// Set up global flag to show exceptions,
// LibPath to search hard disk, and Path
// to search BootDevice and HardDisk
//
GlobalFlag = 8
LibPath = \BootDevice;\SystemDisk\Nt\Dll
Path = \BootDevice;\SystemDisk\Nt\Bin
//
// Start the debug subsystem
//
Debug = \SystemDisk\Nt\SubSys\dbgss.exe
//
// Start the os2 subsystem
//
SubSystem = \SystemDisk\Nt\SubSys\os2ss.exe
//
// Setup Quota
//
PagedPoolLimit = 1048576
NonPagedPoolLimit = 1048576
MinimumWorkingSetSize = 45
MaximumWorkingSetSize = 75
```

2.1 Configuration File Data Structures

The configuration file APIs export several data structures that are visible to it's users.

2.1.1 CONFIG_FILE

The CONFIG_FILE data structure is a "handle" to a configuration file. This data structure is opaque as far as it's users are concerned.

2.1.2 CONFIG_SECTION

The CONFIG_SECTION data structure is a "handle" to a configuration file section. It is used to enumerate and locate keywords within a section. This data structure is opaque as far as it's users are concerned.

2.1.3 CONFIG_KEYWORD

The CONFIG_KEYWORD is the only data structure that is not opaque. This data structure contains strings for the keyword's name and value. It also contains a pointer to the next keyword whose keyword value is the same.

typedef struct **_CONFIG_KEYWORD** {
    STRING *Keyword*;
    STRING *Value*;
    struct_CONFIG_KEYWORD * *NextKeyword*;
    struct_CONFIG_KEYWORD * *LastKeyword*;
} **CONFIG_KEYWORD**, ***PCONFIG_KEYWORD**;

CONFIG_KEYWORD Structure:

Keyword —Supplies the name of the keyword as a counted string. The *Keyword*.Buffer field is a NULL terminated string.

Value —Supplies the value of the keyword as a counted string. If the keyword has no value, then the *Length* and *MaximumLength* fields are zero, and the *Buffer* filed is NULL. Otherwise, *Value*.Buffer is a NULL terminated string.

NextKeyword —Supplies the address of the next keyword having the same keyword name. The end of the list contains a value of NULL. Keywords in this chain are in configuration file order.

LastKeyword —Opaque.

2.1.4 Configuration File APIs

There are several Configuration File APIs.

2.1.4.1 RtlOpenConfigFile

A configuration file is opened using the following API:

NTSTATUS
RtlOpenConfigFile(
    IN PSTRING *ConfigFilePathname*,
    OUT PCONFIG_FILE **ConfigFile*
)

Parameters:

ConfigFilePathname —Supplies the name of the configuration file to open.

ConfigFile —Returns a handle to a configuration file.

Return Value:

SUCCESS() —The configuration file was opened successfully.

!SUCCESS() —A failure occured opening the configuration file, or the configuration file could not be properly parsed.

This function opens the specified configuration file and initializes all associated data structures. Configuration files are broken up into sections, each section contains keywords and associated values.

Sections are identified by a section name enclosed in braces appearing alone on a line. Each section contains zero or more keywords where a keyword is a keyword=value string.

2.1.4.2 RtlCloseConfigFile

VOID
RtlCloseConfigFile(
    IN PCONFIG_FILE *ConfigFile*
)

Parameters:

ConfigFile —Supplies the address of the configuration file to close.

2.1.4.3 RtlLocateSectionConfigFile

PCONFIG_SECTION
RtlLocateSectionConfigFile(
IN PCONFIG_FILE *ConfigFile*,
IN PSTRING *SectionName*
)

Parameters:

ConfigFile —Supplies the address of the configuration file to search for the specified section name.

SectionName —Supplies the section name to locate in the configuration file.

Return Value:

NULL —A matching section name was not found.

NON-NULL —Returns a pointer to the specified section.

This function locates the named section in the specified configuration file.

2.1.4.4 RtlLocateKeywordConfigFile

PCONFIG_KEYWORD
RtlLocateKeywordConfigFile(
IN PCONFIG_SECTION *ConfigSection*,
IN PSTRING *KeywordName*
)

Parameters:

ConfigSection —Supplies the address of the configuration file section to search for the specified keyword name.

KeywordName —Supplies the keyword name to locate in the configuration file section.

Return Value:

NULL —A matching keyword name was not found.

NON-NULL —Returns a pointer to the specified keyword.

This function locates the named keyword in the the specified configuration file section. If multiple values of the same keyword exist, the *NextKeyword* field of the returned keyword points to the list of duplicate keywords.

2.1.4.5 RtlEnumerateSectionConfigFile

PCONFIG_SECTION
RtlEnumerateSectionConfigFile(
IN PCONFIG_FILE *ConfigFile*,
IN BOOLEAN *Restart*
)

Parameters:

ConfigFile —Supplies the address of the configuration file whose sections are to be enumerated.

Restart —Supplies a value that causes the enumeration to start at the beginning of the section list (TRUE), or continue from the last returned section (FALSE).

Return Value:

NULL —All sections have been returned.

NON-NULL —Returns a pointer to the next section.

This function enumerates all of the sections in the specified configuration file. To start at the beginning of the configuration file the *Restart* parameter is specified as TRUE, subsequent sections are returned with a *Restart* value of FALSE. A value of NULL is returned when all of the sections have been returned.

To enumerate the sections in a loop:

```
for(p=RtlEnumerateSectionConfigFile(ConfigFile,TRUE);
p;
p=RtlEnumerateSectionConfigFile(ConfigFile,FALSE) )
```

2.1.4.6 RtlEnumerateKeywordConfigFile

PCONFIG_KEYWORD
RtlEnumerateKeywordConfigFile(
IN PCONFIG_SECTION *ConfigSection*,
IN BOOLEAN *Restart*
)

Parameters:

ConfigSection —Supplies the address of the configuration file section whose keywords are to be enumerated.

Restart —Supplies a value that causes the enumeration to start at the beginning of the keyword list (TRUE), or continue from the last returned keyword (FALSE).

Return Value:

NULL —All keywords have been returned.

NON-NULL —Returns a pointer to the next keyword.

This function enumerates all of the keywords in the specified configuration file section. To start at the beginning of the configuration file section the *Restart* parameter is specified as TRUE, subsequent keywords are returned with a *Restart* value of FALSE. A value of NULL is returned when all of the keywords have been returned.

Keywords having the same name are linked through the *NextKeyword* field. This function does not walk these links. For each keyword returned by this function, the caller must traverse the *NextKeyword* list to enumerate keywords with the same name.

To enumerate the sections in a loop:

```
for(p=RtlEnumerateKeywordConfigFile(ConfigFile,TRUE);
p;
p=RtlEnumerateKeywordConfigFile(ConfigFile,FALSE) )
```