TP3D Software

OS Design Ideas

Cos I was too bored to actually code them…

David Quintana (gigaherz)

Draft

Table of Contents

High-Level stuff

File system structure

The file system path format is based on the standard internet URI format.

The paths are formed like ([] denotes optionality, * denotes “zero or more”):

[<service>:][/<component>]*[/]

Service Roots

Each full path starts with a service root. The service root is an identifier followed by a colon.

Examples:

- “system:” is the main root where all the system devices, services, and other system components lie.
- A partition on a hard drive could be assigned to the “data:” root.
- “ftp:” would be a valid Service Root for a service implementing the File Transfer Protocol.

Directory Components

Each component consists of a forward slash, followed by an identifier. Path components are case-sensitive by default, meaning a component “/Foo” and a component “/foo” do not necessarily point to the same place. Services which choose to implement case-insensitive components should accept any case as input, and should always show lower-case when listing.

Examples:

- “ftp://ftp.mozilla.org/pub/firefox/” could be a perfectly valid path, assuming there’s a “ftp:” service to handle it (the path parsing routines automatically remove extra “/” characters).
- If said “ftp:” handler was installed, the OS would expose a message pipe in “system:/services/ftp” which could be queried for information on the service.

Relative paths: floating, same-relative, and upper-relative

- A path (with or without a service root specifier) which doesn’t start with a forward-slash character is assumed to refer to the current working directory for the specified service (or the current working service if also unspecified).
- A path component with only a single dot as the name refers to the current directory.
- A path component with only two dots (one after the other) as the name refers to the upper-level directory in the tree.

Examples

- “a:/b/./c” is the same as “a:/b/c”
- “a:/b/c/../d” is the same as “a:/b/d”
- “a:/..” is illegal and will either return an error, or resolve to “a:/”
- “a:/b/./././././././../c”, while being redundant, resolves to “a:/c”

The System Root

The system root, “system:” implements most of the service and device interfaces.

system:/		**Description**
	/devices	Lists the devices exposed by the drivers for use in the user side.
	/services	Lists the registered root handler service programs.
	/named	Lists the system-wide named objects.
	/alias	Lists the path-to-root aliases.

User mode API

The user-mode API defines the set of functions implemented in the user-mode system library.

Naming Convention

The goal of the API is to implement a object-oriented interface, while keeping the API independent of specific programming language restrictions.

To achieve this goal, the API will implement the object-oriented paradigm through plain functions and object handles.

API functions names will include the name of the object class they act on at the beginning of the name.

Example

- All functions which work on “file” objects, will have their names start with “File”, like “FileOpen”, “FileRead”, etc.

Function Interface (parameters) Convention

API functions will take the main object handle always as the first parameter (if there’s one), followed by the parameters for the specific function or method. The return value(s) will then follow last, sent as references/pointers, and filled by the function. The actual return value of the API functions will be a result code, telling about the result of the operation, and in case of failure, the details of the error.

The function parameters can have two attributes depending on the usage:

- [IN] means the parameter is passed using the default method for the type.
- [REF] means the parameter is passed as a pointer to a memory location where the data is.

Type Table

	[IN]	[REF]
Any	-	Raw Pointer (void* in C)
Handle	Direct Value (integer)	Pointer to Handle (Handle*)
String	Pointer to Read-only Data (const wchar*)	Pointer to Data (wchar*)
Int/UInt	Direct Value (integer)	Pointer to type (Int*/UInt*)
Struct Types	Pointer to Read-only Data (const <type>*)	Pointer to Data (<type>*)
Buffer/Array	Pointer to Read-only Data (const <type>*)	Pointer to Data (<type>*)

Example

```
ResultCode DirectoryCreateEntry (        // Always return ResultCode
                Handle [IN] dir,         // The first parameter is the object
                String [IN] entryName,   // Then the method parameters
                Handle [IN] file,
                Handle [REF] entry );    // And last a reference to the return handle
```

Result Codes

The result code will be a 32 or 64bit value (depending on the native size of the OS and CPU), and its contents depend on the success or failure status.

A success code is any code with the top-most bit set to 0 (in other words, a positive integer). In case of a success, the rest of the bits in the result code are undefined, and free to use by the function.

In case of a failure, the code will be encoded as follows:

<severity><cause type>[<padding>]<cause code>

<severity> will be a 4bit number with the top-most bit always set to “1”. This leaves severity to a range of 0 through 7, as follows:

- 0. Basic Error: This kind of error should be the most common, and would include invalid parameter errors and similar problems.
- 1. Security Error: This kind of error includes causes derived from missing access privileges.
- 2. Not Available Error: The system cannot execute this function, either because it’s not implemented, or because the resource is unavailable at the moment.
- 3. Fatal Error: This error indicates the resource which the function is corrupted or lost, and the application will have to dispose of the handle and try to re-gain access to it.
- 4 through 7: these error severity codes are internal to the system, and an application should never receive them.

<cause type> will be a 12bit number with system-defined causes (all undefined values are considered reserved and should NOT be used). The value with all bits set to “1” can be used for user-defined result codes where no system code is applicable.

In the 64bit API, <padding> will consist of a series of 32 “0” bits. In the 32bit API, there’s no <padding> present.

<cause code> will be a 16bit number, and its valid values will depend on the <cause type>. As with it, undefined <cause codes> are reserved and should not be used. The exception is with the user-defined cause type, where all cause codes are valid.

API Function Ideas

Named Objects

```
ResultCode ObjectCreateName (
                      Handle [IN] object,
                      String [IN] name );
```

Directory Listing

```
ResultCode DirectoryOpen (
                      String [IN] name,
                      Handle [REF] handle );
ResultCode DirectoryClose ( Handle [IN] dir );
ResultCode DirectoryRewind ( Handle [IN] dir );
ResultCode DirectoryGetCurrent (
                      Handle [IN] dir,
                      Handle [REF] entry );
```

```
ResultCode DirectoryGoNext ( Handle [IN] dir );
ResultCode DirectoryCreateEntry (
                      Handle [IN] dir,
                      String [IN] entryName
                      Handle [IN] file
                      Handle [REF] entry );
ResultCode DirectoryEntryGetName (
                      Handle [IN] entry,
                      String [REF] name );
ResultCode DirectoryEntrySetName (
                      Handle [IN] entry,
                      String [IN] name );
ResultCode DirectoryEntryGetAccessList (
                      Handle [IN] entry,
                      AccessList [REF] list );
ResultCode DirectoryEntrySetAccessList (
                      Handle [IN] entry,
                      AccessList [REF] list );
ResultCode DirectoryEntryDelete ( Handle [IN] entry );
```

File Access

```
ResultCode FileOpen ( String [IN] path, Handle [REF] file );
ResultCode FileClose ( Handle [IN] file );
ResultCode FileGetInformation (
                      Handle [IN] file,
                      FileInformation [REF] info );
ResultCode FileSetInformation
                      Handle [IN] file,
                      FileInformation [REF] info );
ResultCode FileGetStream (
                      Handle [IN] file,
                      String [IN] streamName,
                      Handle [REF] streamHandle );
ResultCode FileCreateStream (
                      Handle [IN] file,
                      String [IN] streamName,
                      Handle [REF] streamHandle );
ResultCode FileDeleteStream (
                      Handle [IN] file,
                      String [IN] streamName );
```

Stream Access

```
ResultCode StreamRead (
                      Handle [IN] stream,
                      UInt [IN] byteSize,
                      Any [REF] buffer );
ResultCode StreamWrite (
                      Handle [IN] stream,
                      UInt [IN] byteSize,
                      Any [REF] buffer );
ResultCode StreamSeek (
                      Handle [IN] stream,
                      UInt64 [IN] position,
                      ReferencePosition [IN] from );
```

Stream Pipes

```
ResultCode StreamPipeCreate ( Handle [REF] pipe );
ResultCode StreamPipeDispose ( Handle [IN] pipe );
ResultCode StreamPipeGetReadStream (
                      Handle [IN] pipe,
                      Handle [REF] stream );
ResultCode StreamPipeGetWriteStream (
                      Handle [IN] pipe,
                      Handle [REF] stream );
```

Message Pipe Access

```
ResultCode MessagePipeCreate
ResultCode MessagePipeDispose
ResultCode MessagePipeSend
ResultCode MessagePipeReceive
ResultCode MessagePipeReceivePending
```

Processes, Modules and Threads

```
ResultCode ProcessCreate
ResultCode ProcessAttachModule
ResultCode ProcessCreateThread
ResultCode ThreadCreate
ResultCode ThreadSetAddress
ResultCode ThreadSuspend
ResultCode ThreadContinue
ResultCode ModuleLoad
ResultCode ModuleUnload
ResultCode ModuleGetExport
```

Memory and Address Mapping

```
ResultCode MemoryAlloc
ResultCode MemoryRelease
ResultCode MemoryMapViewOfMemory
ResultCode MemoryMapViewOfFile
ResultCode MemoryUnmap
```

Structured Exception Handling

```
ResultCode ExceptionPushHandler
ResultCode ExceptionPopHandler
ResultCode ExceptionSetSeverityFilter
ResultCode ExceptionAddResultTypeFilter
ResultCode ExceptionRaise
ResultCode ExceptionRaiseAfterDelay
```

Synchronization

```
ResultCode MutexAlloc
ResultCode MutexDispose
ResultCode MutexAcquire
ResultCode MutexRelease
ResultCode SemaphoreAlloc
ResultCode SemaphoreDispose
ResultCode SemaphoreSetCounter
ResultCode SemaphoreCountDown
ResultCode SemaphoreCountUp
ResultCode WaitableQueueAlloc
ResultCode WaitableQueueDispose
ResultCode WaitableQueueWait
ResultCode WaitableQueueNotify
ResultCode WaitableQueueNotifyAll
```

Base Graphics Layer

```
Resultcode GraphicsDeviceGetCount
ResultCode GraphicsDeviceGetInformation
Resultcode DesktopsGetCount
Resultcode DesktopsGetDesktop
Resultcode DesktopCreate
Resultcode DesktopDestroy
ResultCode DesktopSetParent
ResultCode DesktopGetSize
ResultCode DesktopSetSize
ResultCode DesktopGetSurface
ResultCode DesktopSetSurface
ResultCode SurfaceCreate
ResultCode SurfaceDestroy
ResultCode SurfaceCopy
ResultCode SurfaceGetProperties
ResultCode SurfaceGetImage
ResultCode SurfaceSetImage
ResultCode SurfaceSetBrush
ResultCode SurfaceSetPen
ResultCode DrawLine
ResultCode DrawRectangle
ResultCode DrawEllipse
ResultCode DrawArc
ResultCode DrawPolyline
ResultCode DrawSpline
ResultCode DrawImage
ResultCode DrawImageUnscaled
ResultCode DrawImageBlended
ResultCode DrawImageTransformed
ResultCode FillRectangle
ResultCode FillEllipse
ResultCode FillArc
ResultCode FillPolyline
ResultCode FillSpline
```

More to come…

Mid-level stuff

User-Driver Interface Ideas

Base Driver Model

- Stream Devices
- Random Block Devices
- Message (Variable-size Packet) Devices
- DeviceControl/DeviceRequest

Special-Purpose Driver Interfaces (using DeviceRequest)

- Kernel-mode Service (File System) Handler
- Graphics Device Interface

DeviceControl

The DeviceControl interface function is used to change settings and parameters for a device.

DeviceRequest

The DeviceRequest interface function submits a request to the device’s queue for processing. Once the device is done processing the request, it will mark it as processed and return it to the kernel, which will either unlock the process (

Low-level stuff

Boot Process

The initial steps of the booting process vary depending on the kind of hardware, and the place where the OS boots from.

Some examples are:

- 16bit BIOS boot from Floppy
- 16bit BIOS boot from HDD
- 16bit boot from DOS
- 32bit boot from EFI
- 32bit multiboot loader
- …

Boot sequence

1. (bootloader) Load kernel image from disk
2. (bootloader) Initialize 32bit processor mode (only for 16bit boot modes)
3. (bootloader) Initialize 32bit environment for kernel startup
4. (bootloader) Initialize class 1 drivers and update kernel driver table (known addr.)
5. (bootloader) Jump to kernel base address
6. (kernel) Initialize memory manager (required for class 2 drivers)
7. (kernel) Initialize system timers
8. (kernel) Load and initialize class 2 drivers
9. (kernel) Load and initialize class 3 drivers as the devices are being discovered by class 2 and 3 drivers.
10. (kernel) Initialize task scheduler, event queue, and system service root
11. (kernel) Load user boot process
12. (kernel) Queue user boot task in scheduler
13. (kernel) Tell scheduler to run
14. (kernel) Rest until next syscall

Class 1 Drivers (Bootloader)

These are drivers needed for the first steps of the boot process. They build the bootstrap environment required to load the rest of the kernel and driver modules from the disk. These drivers are configured by the bootloader as part of the initial kernel environment.

- BIOS/EFI keyboard input
- BIOS/EFI console
- BIOS/EFI disk I/O
- Basic Serial-port/Ethernet Debug Interface
- Memory Controller Driver
- Processor Features Driver
- Chipset Features Driver

Class 2 Drivers (Boot Devices)
These are the basic drivers required to initialize the rest of the system, and must not have any dependency on other drivers, but they already use the normal driver system.

- Legacy Keyboard
- Serial port and PS/2 Mouse
- PCI Enumerator
- IDE/AHCI/SCSI Drivers
- OHCI Drivers (USB and FW)
- VGA/VESA Graphics
- Kernel File System Drivers
- …(?)

Class 3 Drivers
Every other driver, even if it doesn't have dependencies, will be a class 3 driver…

Memory Manager
Pages, Access rights, Process ownership, address mapping

Event Queue
Keyboard, mouse, etc.

System Timers
Main timer:

- Fast (10/100khz or so…)
- System counters and timers updated here
- Task scheduler will update every N ticks (configurable).

Task Scheduler
Basic idea: Round-robin with a bunch of priority queues.

Driver Interface

Device Identifiers and Classes
When a driver is loaded, the driver will register a series of filters to tell the kernel which devices IDs or classes it can handle.

Given in some cases a user might want to install a completely virtual device, drivers can request the kernel to generate a virtual device identifier for which the kernel will generate a device descriptor. The driver will then receive that identifier in the handler detection function, where it can expose new devices or device points.

Device Descriptors
After a driver is loaded, the kernel will give it all the devices it can handle. The driver than can accept the device, or reject it.

If the driver accepts the device, the kernel will mark that entry as handled, point it to the driver, and increase the driver's usage count.

If the driver rejects the device, the kernel will try with the next possible driver and if none wants it, mark it as “uncontrolled”. A user-mode tool can then query the device information service and request the user to provide a driver for it (or detect it automatically).

A user-mode process can request the loading of a driver, in which case the kernel will look at all the “uncontrolled” devices if there’s any which match the driver’s filters.

Device Points

When a driver wants to expose a device for use from user-mode processes, it can register a device point. Device points are entries in the system service root, under system:/devices/.

Those device points will then be accessible as any other file, using stream, block, or message interfaces, depending on the device.