Developing with Xbox 360 Unified Storage

By Zsolt Mathe
Software Design Engineer
Advanced Technology Group (ATG)

Published: July 11, 2005
Updated: December 9, 2010

Introduction

This document describes the main features of Xbox 360 Unified Storage (XUS) and the best practices for developing with it. After you read this document, you should have a clear understanding of how Xbox 360 handles storage devices, as well as how it handles saved game data, title data, and downloadable content data.

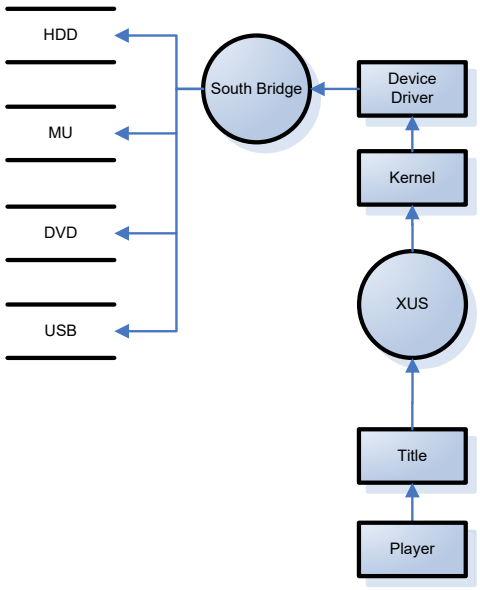
XUS is a flexible programming model that provides an abstraction layer to simplify the handling of multiple storage devices including the hard disk drive, memory units, and other external storage devices.

Because external storage devices may be connected or disconnected at any time, they must be considered transient. XUS can safely handle transient storage devices, and it gives the developer an easy way to handle unexpected removal of such devices.

The XUS Abstraction Layer

The XContent API and the Device Selector component of the Xbox 360 Guide provide storage functionality to the title. Figure 1 illustrates the architecture of the storage hardware.

Figure 1. XUS is an API layer between the kernel and the title



XUS is a robust abstraction layer between the XContent and cache partition APIs. XUS lies between the title and the kernel (see Figure 1). It abstracts how devices are mounted, how titles are partitioned, how storage devices are accessed, and how content is both signed and validated.

The XContent API provides the following functionality:

- Device selection
- Content enumeration
- Assignment and mounting
- Managed content structure
- Signing and validation

Device Selector UI

When the gamer needs to save content to a storage device, Xbox 360 shows the device selector UI. The device selector UI is an Xbox 360 Guide component that shows a list of available storage devices, and allows the gamer to select the storage device that he or she wants to use. The UI automatically shows all available storage devices. The UI will not allow the gamer to select a storage device that is incompatible with the content to be saved.

After the device selector returns successfully, the title is guaranteed that the storage device that the gamer selected is usable and has enough space for the content. Only the title can invoke the device selector UI. It is the title's responsibility to make the device selector available to the gamer through the UI for the title. To ensure a positive user experience, developers need to be aware of some boundary conditions that can occur and how to handle them when calling the device selector UI.

When the title calls **XShowDeviceSelectorUI** and there is only one valid storage device available to the gamer, the system selects that device and does not show the device selector UI.

If the title calls **XShowDeviceSelectorUI** and there is only one valid storage device that doesn't have enough room for the content to be saved, the Device Selector UI will always be shown, and the player will be notified about the space shortage. The title can test for this boundary condition by using **XContentGetDeviceData** and **XContentCalculateSize**.

If there are multiple storage devices and the gamer selects one that doesn't have enough space, the gamer gets an error message, and he or she must choose a different device. To avoid this error, the title should pass the XCONTENTFLAG_MANAGESTORAGE flag when it calls **XshowDeviceSelectorUI**. The size passed to the device selector should be the size of the saved game plus the file system overhead. To calculate the exact space requirement, use the **XContentCalculateSize** API.

To show the device selector UI, first initialize an **XOVERLAPPED** structure, and then call **XShowDeviceSelectorUI** as in the following code example.

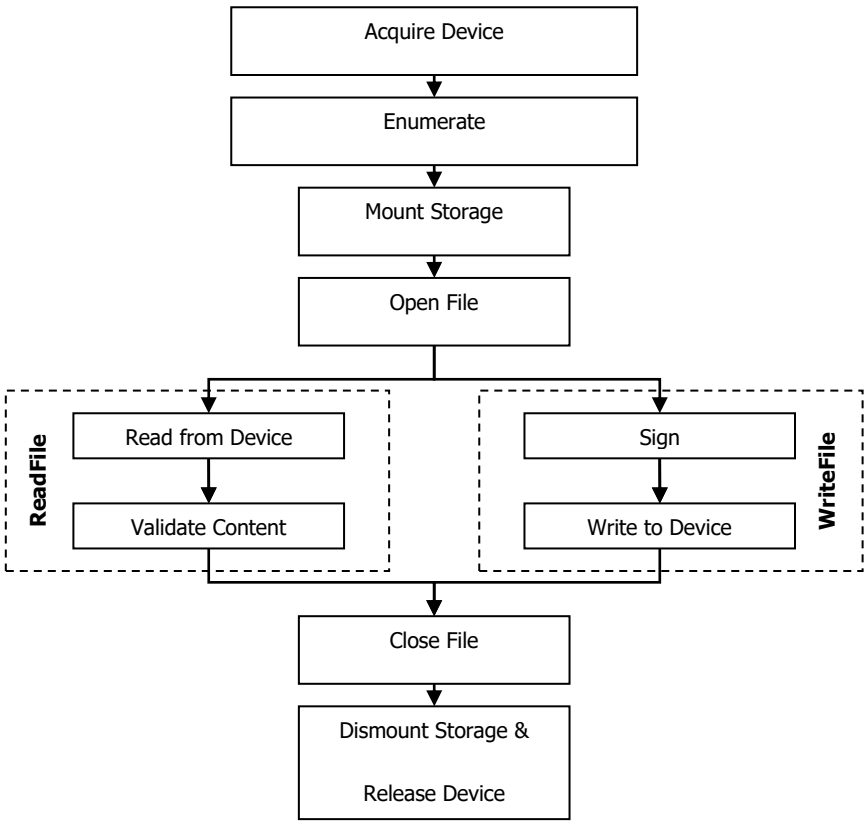
```
XOVERLAPPED xov = {0};
ULARGE_INTEGER iBytesRequested = {0};

// Show device selector UI
XShowDeviceSelectorUI( dwUserIndex,
           XCONTENTTYPE_SAVEDGAME,  // List only saved game
                                    // devices
               0,                   // No special flags
               iBytesRequested,     // Size of data to be written
               &g_DeviceID,         // Return selected device ID
               &xov );
```

When the device has been selected, the overlapped structure is signaled.

Accessing Storage Devices

Whenever you need to access a storage device, whether to read or write data, the process that you follow is well defined. The process consists of acquiring and mounting the device; enumerating content; reading or writing data (at which time the content is signed and/or validated); and dismounting the device. Figure 2 shows the flow of this process.

Figure 2. Process for reading and writing to a storage device



Acquiring and Mounting the Device

First, you need to acquire the device. Almost always, you can acquire a device through the Device Selector UI after the title calls **XShowDeviceSelectorUI**. However, some storage devices, such as hard disk or the game disc, can be acquired automatically without taking any action. After you get the device, you can enumerate the content. Finally, call **XContentCreate** to mount the device. The mount point depends on the type of content to be accessed: saved game content is mounted at a different location than Marketplace content.

Enumerating Content

After you acquire a device, you need to enumerate the content. Content enumeration returns a list of the specified content packages by content type.

When you enumerate content, use a pre-allocated XCONTENT_DATA array. Call **XContentCreateEnumerator** to initialize enumeration. Specify parameters such as the content type, device ID, and the size of the XCONTENT_DATA array. To perform the actual enumeration, call **XEnumerate**:

```
XCONTENT_DATA contentData[CONTENT_DATA_COUNT];
XContentCreateEnumerator( dwUserIndex,
```

```
                                g_DeviceID,
                                XCONTENTTYPE_SAVEDGAME,
                                0,
                                ARRAYSIZE( contentData ),
                                &cbBuffer,
                                &hEnum );

DWORD dwReturnCount;
if( XEnumerate( hEnum, contentData, sizeof( contentData ),
                    &dwReturnCount, &overlapped ) ==
                  ERROR_SUCCESS )
{
    // Do work here
}

CloseHandle( hEnum );
```

Writing and Reading Content

After you enumerate the content, you can write it to the storage device. When writing or reading content, the title uses either the Win32/XAPI file API or the C-runtime file API to perform typical file I/O operations. The following code snippet shows you how to properly write saved game content.

First, the device that is selected by the device selector UI with **g_DeviceID** is mounted to a location instance "My Saved Game 1" with a custom drive name (for more information, see "Dynamic Drive Names" later in this paper).

Next, the content container file specified in the **szFileName** field is created. The container file name length must not exceed 40 characters.

Next, call **XContentCreate** to open a content location for writing. When writing content, pass XCONTENTFLAG_CREATEALWAYS to the function; when reading content, pass XCONTENTFLAG_OPENEXISTING. **XContentCreateEx** provides extra functionality such as allowing for pre-sizing of the container and specifying the file cache size. The XContent API has other flags that specify whether you can transfer content to another user or another storage device and whether content requires strong signing. For a complete list of flags, see the XDK documentation.

Next, the standard APIs **CreateFile** and **WriteFile** are used to write a file of arbitrary data. This example shows saved game content. However, other types of content are accessed in the same way.

```
// Initialize content type and device struct
XCONTENT_DATA contentData = {0};
wcscpy_s( contentData.szDisplayName, L"My Saved Game 1" );
strcpy_s( contentData.szFileName, "content_file.bin" );
contentData.dwContentType = XCONTENTTYPE_SAVEDGAME;
contentData.DeviceID = g_DeviceID;  // Device ID returned from
                                    // XShowDeviceSelectorUI

// Mount the device to the dynamic drive name "savedrive"
XContentCreate( dwUserIndex, "savedrive", &contentData,
XCONTENTFLAG_CREATEALWAYS, NULL, NULL, NULL );

// Create a saved game file
```

```
HANDLE hFile = CreateFile( "savedrive:\\savegame.bin",
                  GENERIC_WRITE, 0, NULL, CREATE_NEW,
                  FILE_ATTRIBUTE_NORMAL, NULL );
if( !WriteFile( hFile, pBuffer, dwBytes, &dwWritten, &overlapped
) )
{
      // Check GetLastError() to determine if there is no
      // space, corruption or other
}
CloseHandle( hFile );

XContentClose( "savedrive", NULL );
```

When reading content, in order to be TCR compliant, you must call **XContentGetCreator** before you call **XContentCreate**. **XContentGetCreator** checks whether the user who is signed in currently is the same user who created the content package. If the returned value is FALSE, the title won't load the saved game. The following code snippet shows you how to read saved game content.

```
// Initialize content type and device struct
XCONTENT_DATA contentData = {0};
wcscpy_s( contentData.szDisplayName, L"My Saved Game 1" );
strcpy_s( contentData.szFileName, "content_file.bin" );
contentData.dwContentType = XCONTENTTYPE_SAVEDGAME;
contentData.DeviceID = g_DeviceID;  // Device ID returned from
                                    // XShowDeviceSelectorUI

XContentGetCreator( dwUserIndex, &contentData, &bUserIsCreator,
&xuid, NULL );

if( bUserIsCreator == FALSE )
      return TITLE_ERROR_CANT_READ_CONTENT;

// Mount the device to the dynamic drive name "savedrive"
XContentCreate( dwUserIndex, "savedrive", &contentData,
XCONTENTFLAG_OPENEXISTING, NULL, NULL, NULL );

// Create a saved game file
HANDLE hFile = CreateFile( "savedrive:\\savegame.bin",
                  GENERIC_READ, 0, NULL, OPEN_EXISTING,
                  FILE_ATTRIBUTE_NORMAL, NULL );
if( !ReadFile( hFile, pBuffer, dwBytes, &dwRead, &overlapped ) )
{
      // Check GetLastError() to determine if there is no
      // space, corruption or other
}
CloseHandle( hFile );

XContentClose( "savedrive", NULL );
```

Content Signing and Validation

All content accessed through the XContent APIs is signed and validated. Content is signed automatically when you use the file I/O APIs (such as **WriteFile** or **fwrite**) to write data. Content is validated automatically when you use the XContent APIs (such as **XContentCreate**, **ReadFile**, or **fread**) to read data.

When content is signed, a hashing method is used to generate a content signature. The content signature is finalized when the file handle is closed. When content is validated, both the signature and licensing restrictions of the content are checked. Different content types are validated differently: saved games require only signature validation while Marketplace content may have per-console or per-user licensing restrictions.

Because content is always validated, games don't need to worry about read operations returning invalid data. If there is corruption in the header of the content package (the first 0xA000 bytes), **XContentCreate** may return an ERROR_FILE_CORRUPT error code. **CreateFile** and **ReadFile** return ERROR_DISK_CORRUPT if there is corruption within the container file system other than its header (after the first 0xA000 bytes).

The performance impact of signing and validation is minimal when accessing data in blocks of at least 4 KB.

Dealing with Transient Devices

If the gamer removes an external storage device that has been mounted by the system, the system automatically prompts the gamer to reinsert the device. Whenever a device is connected or disconnected, the XContent API sends the XN_SYS_STORAGEDEVICESCHANGED event.

To find out if a device is attached or not, call the **XContentGetDeviceState** API, passing it the device ID. If the device is not attached, the error code ERROR_DEVICE_NOT_CONNECTED is returned from XContent and Win32 file I/O APIs. If this error code is returned, all file and XContent handles must be considered invalid and closed.

When the system receives an event indicating that a storage device was inserted or removed, the title must re-enumerate available content locations by calling **XContentCreateEnumerator**. To enumerate content on all connected devices, pass XCONTENTDEVICE_ANY as the device ID.

Content Containers

The Xbox 360 storage system organizes content based on user, publisher, title, and content type. The content data is stored in a container file specified by the **szFileName** field in the XCONTENT_DATA structure. The container file is stored in a directory structure that is denominated by publisher, title, and user. Within the container, there is a file system (STFS) that manages the content and takes care of security.

Because of the structure of the file system, if you delete a file from a mounted location, the space taken by the file is not automatically reclaimed until **XContentClose** or **XContentFlush** is called.

Logical Drive Names

Logical drive names are built-in system drive names that are accessed directly with standard file I/O APIs. Dynamic drive names are created with **XContentCreate**. Logical drive names are reserved, and must not conflict with any dynamic drive names created by the title.

Logical drive names can be 1–12 characters in length. Logical drives can have more than one name that is descriptive and another name that consists of a single character (such as "a" or "d"). Although you can access logical drives by their single character name, we recommend that you access the drives by their descriptive names instead. As new devices are added in the future, they will be named with the descriptive drive name only. For backward compatibility, single-letter drive name equivalents are still supported for the game disc drive. The following table shows drive names that are available on Xbox 360. You can use standard file I/O APIs to access the following logical drive names.

Drive Name	Availability	Description
devkit:\ e:\	Dev Kits Only	Root of the development partition on devkits. To enable access to this partition, call **DmMapDevKitDrive** before performing any I/O operations. Retail titles are not allowed to access the e:\ partition directly.
game:\ d:\	Retail and Dev Kits	Root of the game disc or emulated game disk.
cache:\	Retail and Dev Kits	Scratch partition allocated by the system to the game. The cache partition is always stored on the retail hard drive and is not available if the hard drive is not attached. The size of the cache partition available to the title is 2GB.

All other devices must be accessed through the XContent API.

Dynamic Drive Names

Title developers address storage devices through dynamically-mapped logical drive names. These dynamically-mapped drives will always point to the appropriate user storage container on the selected device. Using this model, the title sees the container as a file system as XUS manages the low-level interaction with the device.

The developer can choose any symbolic name other than the reserved logical drive names. The name can be from 1 to 12 characters in length. The title calls

XContentCreate to map the symbolic drive name, and then uses that drive name with the system functions **CreateFile**, **ReadFile**, and **WriteFile** to create, read, and write content data to the storage location.

The Cache Partition

The cache partition on the Xbox 360 provides 2 GB of user accessible storage. You can use it to cache game assets, to speed up loading assets from the DVD, or to provide temporary storage.

Internally, the cache partition is very similar to content packages. The cache partition file system is contained within a single file, and uses the same layout (STFS) as the file system in a content package. You only need to mount the cache partition once with **XMountUtilityDrive**. Because the file system is transactional, data written to the partition is not committed until you call **XFlushUtilityDrive**. For this reason, you should call **XFlushUtilityDrive** frequently to avoid data loss due to external factors.

You can use **XFileCacheInit** to perform automatic file caching from the game disc. The system copies the files from the game disc to the cache partition in the background, whenever the title is not accessing the game disc. If automatic file caching is enabled, you don't need to make any more changes to the title code. You can control automatic file caching by preloading specific files, by turning opportunistic caching on or off, and by using callbacks. For more information, see the white paper Effective Use of Game Disc File Caching.

If you use the cache partition, be sure to keep the file count lower than 1,000 and the file sizes below 100 MB. Because of the secure nature of the STFS file system, large file operations may take up to a few seconds.

Best Practices

- To ensure high performance, call **XContentCreate** and **XContentClose** as few times as possible. These APIs allocate resources and perform content verification that has fairly high overhead.
- Prefer threading or asynchronous versions of both XContent APIs and file I/O APIs. Doing so ensures that your title will perform optimally with slow devices, as well as future connected devices.
- Avoid reading or writing data in small blocks. Not only does content signing require an extra write operation, but small I/O operations result in lower performance because of driver and device overhead.
- For maximum performance, use 4 KB aligned buffers that are a multiple of 4 KB in size.
- Consider using fewer, larger files, rather than many small files.

- To determine how much free space is available, use the **XContentGetDeviceData** API after the device has been mounted with **XContentCreate**. **XContentGetDeviceData** does not take into consideration the STFS file system overhead. You should use **XContentCalculateSize** to account for file system overhead. To allow the gamer to select devices with enough free space, specify how much space you need when showing the device selector UI.
- Prefer the native file I/O APIs because C/C++ file I/O APIs provide limited error handling and have extra overhead.
- Listen to the device removal and insertion events to determine if the currently mounted device is valid. Do not rely solely on error codes from the file I/O APIs to determine the connection status of a storage device.
- Whenever possible, pre-size the content container by calling **XContentCreateEx** if you know the maximum amount of space required.
- Check for ERROR_DISK_CORRUPT and ERROR_FILE_CORRUPT errors from all XContent and file I/O APIs, and show messages to the player, when appropriate.
- Call **XContentFlush** for content packages and **XFlushUtilityDrive** for the cache partition often enough to not lose data if the device is removed or if the console loses power. Be sure to check the return values to make sure that all data is committed to the partition.

Summary

Xbox 360 Unified Storage provides the means for titles to easily access a wide range of storage devices. XUS is easily integrated into titles and it allows the use of existing standard file I/O functions. For more example code, see the XContent sample located in the Source\Samples\System directory of the Xbox 360 SDK.