XboxConsole

Getting Started Guide

Contents

[Introduction 3](#_Toc405276836)

[Overview 3](#_Toc405276837)

[Prerequisites 4](#_Toc405276838)

[Best practices 5](#_Toc405276839)

[How to connect to the Xbox One console 6](#_Toc405276840)

[How to reboot and shutdown an Xbox One console 7](#_Toc405276841)

[How to manage Xbox One configuration settings 8](#_Toc405276842)

[How to manage users on the Xbox One 10](#_Toc405276843)

[Listing users 10](#_Toc405276844)

[Adding users 10](#_Toc405276845)

[Deleting users 11](#_Toc405276846)

[Signing in users 11](#_Toc405276847)

[Signing out users 12](#_Toc405276848)

[Creating parties and adding local users 12](#_Toc405276849)

[Retrieving unique party identifier 13](#_Toc405276850)

[Retrieving party members 13](#_Toc405276851)

[Removing local users from party 13](#_Toc405276852)

[Inviting remote users to a party 14](#_Toc405276853)

[Joining a remote party 14](#_Toc405276854)

[The lifetime of a party 14](#_Toc405276855)

[How to take a screenshot using XboxConsole 17](#_Toc405276856)

[How to record a DVR capture using XboxConsole 17](#_Toc405276857)

[How to push deploy an application to the Xbox One Console 18](#_Toc405276858)

[Progress Reporting 19](#_Toc405276859)

[Deployment Cancellation 20](#_Toc405276860)

[How to register a package located on TitleScratch drive 20](#_Toc405276861)

[How to get available space for application installation 20](#_Toc405276862)

[How to control the lifecycle of a package on the Xbox One console 21](#_Toc405276863)

[Separation of functionality 21](#_Toc405276864)

[Registering an event handler for the ExecutionStateChanged event 22](#_Toc405276865)

[How to uninstall a package 23](#_Toc405276866)

[How to launch a non-package executable 23](#_Toc405276867)

[How to copy files and directories 24](#_Toc405276868)

[Copying files to and from the System OS 24](#_Toc405276869)

[Copying files to and from the Title OS 27](#_Toc405276870)

[Copying files to and from an application container 27](#_Toc405276871)

[Copying files to and from the @scratch drive 28](#_Toc405276872)

[Copying directories to and from the Xbox One console 28](#_Toc405276873)

[Getting progress updates while copying 29](#_Toc405276874)

[How to capture debug output from the Xbox One console 30](#_Toc405276875)

[How to simulate controller inputs 31](#_Toc405276876)

[How to pair a controller with a user 32](#_Toc405276877)

[Exception handling 33](#_Toc405276878)

[Remarks 33](#_Toc405276879)

# Introduction

This documentation is available to help you understand what the **XboxConsole** library is and how to use it in your testing processes. This document will provide a high level overview of the main features in the **XboxConsole** library. For full details about the specific classes and members, see the .chm file.

# Overview

The **XboxConsole** provides a managed wrapper for the XDK that allows you access and control over the Xbox One console for automation and other customized tooling scenarios. The **XboxConsole** library exposes the following objects.

* **XboxConsole**: This class gives you access to core components of the console. With instances of this class, you can get a list of installed packages on the console, see all running processes, receive debug output strings, shutdown, reboot, etc…
* **XboxFile:** This class provides static methods for copying, moving, and deleting of files from the console.
* **XboxFileInfo**: This class provides properties and instance methods for copying and deleting of files from the console.
* **XboxDirectory**: This class provides static methods for creating, copying, moving, and deleting of directories from the console.
* **XboxDirectoryInfo**: This class provides properties and instance methods for creating, copying, and deleting of directories from the console.
* **XboxUser**: This class provides methods for signing a user in or out, pairing with a controller, and creating or joining parties.
* **XboxParty**: This class provides functionality for getting a unique party identifier used to accept party invitations on a console other than the one on which the invitations were sent, as well as retrieving party members.
* **XboxRemoteUser**: This class is used to invite users on another console using their XUID (unique Xbox Live identifier). XUID of local users can be retrieved using the XUID property of **XboxUser** class, which is valid only if a user is signed in.
* **XboxRemoteParty**: This class is used to refer to a party created on another console by its **PartyId** while a user on console that did not create this party is accepting or declining a party invitation.
* **XboxGamepad:** This object allows you to simulate user control inputs.
* **XboxConfiguration**: This object allows you to get or set various configuration settings on the console.

The classes listed above are your primary entry points into the **XboxConsole** library. With these objects, you can programmatically control an Xbox console and simulate actions that you would otherwise have to manually perform using the XDK’s command lines or a physical Xbox controller.

**Notes**:

* All of the methods in **XboxFile** and **XboxDirectory** are static, it might be more efficient to use these static methods if you want to perform only one action. However, if you intent to reuse an object, consider using the instance methods of an **XboxFileInfo** or **XboxDirectoryInfo** instead.
* To track file IO on the console, you will use the **XboxFileInfo** object; however, to track file IO on your PC, you will use the **FileInfo** object from the .Net library. See also [System.IO Namespace](http://msdn.microsoft.com/en-us/library/system.io.aspx).
* XboxConsole can be configured to automatically log telemetry based on user activity. The following events can be logged with the corresponding information:

|  |  |
| --- | --- |
| Event | Data Logged |
| Module Loaded | * User Name * Machine Name * Tool Executable Name |
| Method Called | * Method Signature (no parameter values are logged) |
| Object Created | * Constructor signature |

To enable, you will need to configure a SQL endpoint.

* Create a [SLAB](https://slab.codeplex.com/) SQL sink using the scripts provided in the [SemanticLogging.Database NuGet package](http://www.nuget.org/packages/EnterpriseLibrary.SemanticLogging/).
* Set the **LoggingConnectString** const string in **Microsoft.Internal.Xbox.GamesTest.Telemetry.TelemetrySink** to the connection string for the endpoint you created.

# Prerequisites

Before you get started, make sure you have the following items ready.

* Visual Studio 2012.
* Download or build the latest binaries and add appropriate references to your project.

References to add:

* + GamesTest.XboxConsole.dll
  + GamesTest.XboxConsole.Infrastructure.dll
  + Adapter libraries appropriate for the desired XDK versions
* You must have a 64-bit machine to build and run the **XboxConsole** library because it takes a dependence on the 64-bit binaries included in the XDK.
* While the XDK is not currently required at compile-time, it is required to be installed on the PC at runtime. Make sure you have the latest release of the XDK installed on your PC.
  + Make sure your Xbox console is flashed to the same version of the XDK installed on the PC. Run corresponding **Recovery Update**, if needed.
  + Additionally, the **XboxConsole** library also supports having the ADK installed in lieu of, or in addition to, the XDK.
* Know your console’s **System IP** address so you can connect to it. You may also run the **xbconnect** command to set a default console, then your test tools do not need to explicitly specify the IP every time you need to connect to the console.
  + **Note:** you must always use the System IP for connecting to the console when using XboxConsole API, even when setting the default console using xbconnect. Setting the default console using its Tools IP address will cause XboxConsole’s default constructor to initialize with Tools IP, and the System IP property will report Tools IP rather than System IP. XboxConsole functionality will not fail due to this, but it may cause failure in other tools or libraries that rely on getting the System IP.
* Make sure you provisioned your Xbox console. Run the **xbprovision** command using the Xbox One command prompt if needed. Be sure a default console is set on your PC.

# Best practices

|  |  |
| --- | --- |
| Description | Suggestions |
| Should I keep an XboxConsole object alive for reuse or should I create a new instance each time? | It depends on your usage scenario.  That said, it doesn’t matter how long you should keep the object alive. However, **XboxConsole** is a Disposable object, the recommended usage of this object type is by using the **using** syntax:  using(XboxConsole c = new XboxConsole())  {  …  }  This syntax calls the **Dispose** method at the end for you automatically. So, if you are not using the **using** syntax, you must explicitly call the **Dispose** method.  For most use cases, it is a good idea to use the **using** syntax except for when you need the object alive longer. Such as the case when waiting to receive debug output. When receiving debug output, the XboxConsole object must not be disposed until you are done receiving debug output.  Also, whether you are using the **using** syntax or not, we recommend that you only instantiate one instance of this object per console. |
| Should I check for valid object on the XboxConsole object before using it? | No. All relevant validation takes place in the constructor so it is ready for use after it is instantiated.  However, you may want to wrap certain method calls in a try/catch block to catch any exceptions that are thrown there. |
| What are some exceptions to handle? | For info on exceptions, see **Exception handling** section. |

# How to connect to the Xbox One console

To use the **XboxConsole** library you must know your console’s System IP address or host name, and its session key. With this information, you can connect to the console in one of two ways: (1) you can connect to a console by explicitly instantiating the object with the console’s IP address/host name and session key. Or, (2) you can set a default console on your machine and instantiate an **XboxConsole** object using the default constructor.

**Note:** the Host Name resolution will only work when the PC connecting to the Xbox is on the same subnet (the third quadrant of the IP address) as the Xbox. When they are on different subnets, a “no such host is known” exception will be raised.

To set the default console on your machine, open an Xbox One command prompt and run the following command:

xbconnect <your console’s System IP address>

Or

xbconnect <your console’s host name>

Or

xbconnect <your console’s System IP address>+<your console’s session key>

Or

xbconnect <your console’s host name>+<your console’s session key>

The **xbconnect** command takes an IP address of the console and, if your console has one set up, the session key. This command will register this IP as the default console. Then, in your test tool, you can instantiate a new **XboxConsole** object by calling its default constructor. Like this:

XboxConsole xbc = new XboxConsole();

**Note:** The System IP address must be passed to xbconnect to set the default console for use with XboxConsole’s default constructor. Passing Tools IP will cause XboxConsole to report Tools IP from the SystemIPAddress property.

If you want to target a specific console, you can instantiate an **XboxConsole** and give it the IP address of the console you want to connect to. For example:

XboxConsole xbc = new XboxConsole(IPAddress.Parse("<your console’s System IP address>"));

Or if the console has a session key:

XboxConsole xbc = new XboxConsole(IPAddress.Parse("<your console’s System IP address>"), "<your console’s session key>");

To initialize with host name, use the “connection string” constructor, which also supports IP addresses:

XboxConsole xbc = new XboxConsole("<your console’s host name or System IP address>");

XboxConsole Configuration (accessed using its **Configuration** property) can be used to retrieve the Host Name later, even if it was initialized with an IP address.

The session key can be passed into the same constructor by separating it from host name/IP address using a ‘+’ character, same as with xbconnect command line tool:

XboxConsole xbc = new XboxConsole(

"<your console’s host name or System IP address>+<session key>"

);

After initializing XboxConsole, you can use the **ConnectionString** property to get System IP address/host name and session key it was originally initialized with, in one string (with IP/host separated from session key using the ‘+’ character). This functionality can also be used to clone a connected instance of XboxConsole, by initializing a second XboxConsole and passing a ConnectionString from the first XboxConsole into its constructor:

XboxConsole second = new XboxConsole(first.ConnectionString);

This also opens the scenario where you may want to control multiple consoles from the same test tool. For example, you might want to reboot all your consoles at the same time by iterating through a list of IP addresses and issuing the **Reboot** command on each console.

# How to reboot and shutdown an Xbox One console

To programmatically reboot and shutdown an Xbox console, you will need to instantiate an **XboxConsole** object. Then you can use that object to call these two operations. The following code snippet illustrates how to **Reboot** your default console. You can call **Shutdown** in the same way.

using (XboxConsole xbc = new XboxConsole())

{

…

xbc.Reboot();

}

This will reboot the console and wait indefinitely for the console to finish rebooting. If you would rather not wait that long, you can set a timespan for this call to wait. If it does not complete rebooting within this timespan, it will throw a **TimeoutException** exception. To set a timeout, do the following instead.

xbc.Reboot(TimeSpan.FromMinutes(2));

# How to manage Xbox One configuration settings

The XboxConsole library provides API that allows you to get and set Xbox One configuration settings programmatically instead of using the **xbconfig** command line utility.

You can read settings using **Configuration** property of **XboxConsole** object. The property returns an object that implements **IXboxConfiguration** interface and contains read-only properties (like **SandboxId**), one for each of the console’s settings.

You can change the console’s settings by creating a new **XboxConfiguration** object, either empty (using the default constructor) or filled with settings from an already existing configuration (using a constructor that takes **IXboxConfiguration** as a parameter). **XboxConfiguration** object provides read-write properties that allow you to set new values for the console’s settings.

Any settings left unaltered will remain **null**; all **null** settings will not be changed on the console upon reboot.

A console needs to be rebooted to make new setting values to become available to titles and apps running on the console. This is true regardless of whether you are using only **xbconfig** or the **XboxConsole** library. The **XboxConsole** object provides a **Reboot** method that takes an **XboxConfiguration** object as a parameter to apply the setting values it contains.

If **XboxConfiguration** object was created using the default constructor then only those settings will be applied that were set by user. If **XboxConfiguration** object was created using the constructor that takes **IXboxConfiguration** object as a parameter then all settings will be applied regardless of whether they were or were not modified by user after that.

Real use example:

using (XboxConsole console = new XboxConsole(IPAddress.Parse("<console IP>")))

{

// -------- How to get configuration settings -------

// Returns an object which implements IXboxConfiguration

IXboxConfiguration readOnlyConfiguration = console.Configuration;

// Returns SandboxId setting value

string sandboxID = readOnlyConfiguration.SandboxId;

// readOnlyConfiguration.SandboxId = null; // won't compile - the property is read-only

// -------- How to set configuration settings --------

XboxConfiguration myConfiguration = new XboxConfiguration(console.Configuration); // Creates a copy of configuration that allows to change settings.

// It is also possible to create a configuration using the default constructor.

// In this case, only those settings that have been explicitly set will be applied.

myConfiguration.SandboxId = "<new Sandbox ID>";

console.Reboot(myConfiguration);

}

The **XboxConfiguration** object exposes the following settings:

* **Environment** – This configuration setting is represented as a **string**. This setting may be disabled; altering this setting could potentially send the console into an unstable state.
* **SandboxId** – This configuration setting is represented as a **string**.
* **OOBECompleted** – This configuration setting is represented as a **Boolean** value. This setting will only return “false” when the Out of Box Experience (OOBE) remains incomplete; setting this value to “true” will force the user to repeat the OOBE.
* **ProfilingMode** – This configuration setting is represented as a **Boolean** value.
* **PreferredLanguages** – This configuration setting is represented as an **IEnumerable<CultureInfo>** collection. This setting does not support custom **CultureInfo** definitions, nor does it support cultures existing outside of **CultureTypes.SpecificCultures** collection.
* **GeographicRegion** – This configuration setting is represented as a **RegionInfo**. This setting does not support custom **RegionInfo** definitions, nor does it support neutral region definitions.
* **TimeZone** – This configuration setting is represented as a **TimeZoneInfo**. This setting does not support custom **TimeZoneInfo** definitions.
* **ConnectedStorageForceOffline** – This configuration setting is represented as a **Boolean** value.
* **SimulateVersionSwitch** – This configuration setting is represented as a **Boolean** value.
* **EnableKernelDebugging** – This configuration setting is represented as a **Boolean** value.
* **SessionKey** – This configuration setting is represented as a **String** value. This setting must be an alphanumeric string of length 31 or less. Set this value to string.Empty to clear the SessionKey (passing null will cause this setting not to be applied).
* **HostName** – This configuration setting is represented as a String value. This setting must be an alphanumeric string of no more than 15 characters that doesn’t start with a number and contains no spaces. HostName cannot be string.Empty or null.
* **AccessoryFlags** – This configuration setting is represented as a **UInt32** value (uint).
* **PowerMode** – This configuration setting is represented as a **PowerModeType** enumeration.
* **IdleShutdownTimeout** – This configuration setting is represented as an **IdleShutdownTimeoutType** enumeration because it supports only three values.
* **DimTimeout** – This configuration setting is represented as an **int** value with a valid range of 0-255 in increments of 5.
* **HttpProxyHost** – This configuration setting is represented as a **String** value and must be set to a valid IP address followed by a colon and a port name. Set to string.Empty to clear (passing null will cause this setting not to be applied).
* **DisplayResolution** – This configuration setting is represented as a **DisplayResolutionType** enumeration.
* **ColorSpace** – This configuration setting is represented as a **ColorSpaceType** enumeration.
* **ColorDepth** – This setting is represented as an **int** with valid values of 24, 30 and 36.
* **NetworkType** – This configuration setting is represented as a **NetworkTypeType** enumeration, and is read-only.
* **NetworkAddressMode** – This configuration setting is represented as a **NetworkAddressMode** enumeration, and is read-only.
* **DefaultUser** – This configuration setting is represented as a **String** value containing a valid user account email address that matches one of the accounts added to the kit. This will auto-login the specified user at kit startup. For successful auto-login, the user must have been previously added to the kit with “save password” option. Set to string.Empty to clear (passing null will cause this setting not to be applied).
* **DefaultUserPairing** – This setting is represented as a **UserPairingType** enumeration.
* **WirelessRadioSettings** – This configuration setting is represented as a **WirelessRadioSettingsType** enumeration.
* **HdmiAudio** – This configuration setting is represented as a **HdmiAudioOutput** enumeration.
* **OpticalAudio** – This setting is represented as an **OpticalAudioOutput** enumeration.
* **AudioBitstreamFormat** – This configuration setting is represented as an **AudioBitstreamFormatType** enumeration.

# How to manage users on the Xbox One

XboxConsole allows programmers to manage users on the Xbox One. You can list, add, delete, sign-in, and sign-out users on the console. XboxConsole provides an **XboxUser** type which can be used to view the Email address, gamertag, user id, and a flag indicating whether or not the user is signed in.

## Listing users

Users are listed using the **Users** property of an **XboxConsole** object.

using (XboxConsole xbc = new XboxConsole())

{

IEnumerable<XboxUser> users = xbc.Users;

// Use the user enumerable

}

## Adding users

Adding users can be done using the **AddUsers** method of an **XboxConsole** object.

using (XboxConsole xbc = new XboxConsole())

{

XboxUser user = xbc.AddUser(@"UserEmailAddress@xboxtest.com");

// Use the added user.

}

Guest users can also be added using the **AddGuestUser** method while there is a signed in user on the console. If no users are signed in, **AddGuestUser** will fail with a “Failed to add guest user” **XboxConsoleException** built on top of “Unknown COM Error” inner exception.

using (XboxConsole xbc = new XboxConsole())

{

XboxUser user = xbc.Users.First(x => x.EmailAddress == @"UserEmailAddress@xboxtest.com");

user.SignIn("testPassword”);

uint guestUserId = xbc.AddGuestUser();

}

**Notes:**

* The **GamerTag** and **Xuid** properties won’t be populated until an added user has been signed in at least once on the console. Attempting to access Xuid when a user is not signed in will result in XboxUserNotSignedIn exception.

## Deleting users

Specific users can be deleted using the **DeleteUser** method.

using (XboxConsole xbc = new XboxConsole())

{

XboxUser user = xbc.Users.First(x => x.EmailAddress == @"UserEmailAddress@xboxtest.com");

xbc.DeleteUser(user);

}

Deleted users are signed out of the console. If the users being deleted belong to one or more parties, they will be removed from those parties. Parties that lose all of their members will get disbanded and Party ID for those parties will no longer be valid.

Another option to delete users on the console is using the **DeleteAllUsers** method to remove all users on the console.

using (XboxConsole xbc = new XboxConsole())

{

xbc.DeleteAllUsers();

}

## Signing in users

Users are signed in using the **SignIn** method of an **XboxUser** object.

XboxUser user = xbc.Users.FirstOrDefault();

user.SignIn("TestPassword”, false);

The second parameter of the SignIn method is a Boolean value that specifies whether or not to store the password on the console. If a password has been stored on the console, the user may pass-in null for the password.

**Notes:**

* In June 2014 release of the XDK, the password text is not encrypted when it is passed over the network to the console by **SignIn**. If this is a concern in your network environment, then use the Sign-In app on the console to sign in the user and store the user password. (The Sign-In app sends the password over the network encrypted.) After this, you can subsequently use **SignIn** to sign in without passing a password.

## Signing out users

Users are signed in using the **SignOut** method of an **XboxUser** object.

XboxUser user = xbc.Users.FirstOrDefault();

// Sign in and do work with the user.

user.SignOut();

**PairWithVirtualController** method (taking an XboxGamepad object) and with physical controllers using **PairWithPhysicalController** method. For more information and code samples see [How to pair a controller with a user](#_How_to_pair) section.

## Creating parties and adding local users

A new party can be created using **AddLocalUsersToParty** method. The **XboxUser** that calls the **AddLocalUsersToParty** method will be the acting user responsible for creating the party. This method accepts zero or more local users to add to the party, and when the party is created these users will be listed as party members in addition to the party creator. The reference to a local party associated with a specific title to pass into this function can be retrieved using XboxParty.FromTitleId:

// Choose the party creator

XboxUser partyCreator = this.Console.Users.FirstOrDefault();

// Choose 5 local users to add to the party.

// Make sure we don’t include the party creator

// (compare using User ID – cannot compare using Xuid because Xuid is only valid after users are signed in).

XboxUser[] addUsers = (from xu in this.Console.Users where xu.UserId != partyCreator.UserId select xu).Take(5).ToArray();

// All users involved must be signed in for the party to be created

partyCreator.SignIn("password", false);

foreach (XboxUser user in addUsers)

{

user.SignIn("password", false);

}

// Create a local party with above users

XboxParty party = XboxParty.FromTitleId(this.Console, TestTitleId);

partyCreator.AddLocalUsersToParty(party, addUsers);

To create a party with just one member (the creator), pass a zero-length array to AddLocalUsersToParty:

// Create a local party with one member

XboxParty party = XboxParty.FromTitleId(this.Console, TestTitleId);

partyCreator.AddLocalUsersToParty(party, new XboxUser[0]);

**Notes:**

* All local users being added to the party, including the party creator, must be signed in for the party to be created. If one or more users are not signed in, **AddLocalUsersToParty** will throw **XboxUserNotSignedInException** which specifies the email address of the first user that wasn’t signed in.
* If any of the users that are passed in cannot be included into the party (for instance if the party creator is also passed in for “users to add” parameter), an **InvalidXuidException** will be thrown.
* A user can create multiple parties and a user can join multiple parties.

## Retrieving unique party identifier

After a party has been created, its unique identifier (used to accept or reject invitations to this party on another console) can be retrieved using the **PartyId** property:

string partyId = party.PartyId;

## Retrieving party members

Party members of a local party represented by **XboxParty** can be retrieved using the **Members** property. This property returns an IEnumerable of **XboxUserBase** objects which could be **XboxUser** (representing local party members) or **XboxRemoteUser** (representing users on other consoles who were invited to the party).

## Removing local users from party

The **RemoveLocalUsersFromParty** method can be used to remove previously added local users from a party by passing in a list of users to remove:

partyCreator.RemoveLocalUsersFromParty(party, previouslyAddedUsers);

Or, use the **Members** property to remove all local users from the party (including only XboxUser objects and excluding XboxRemoteUser objects):

partyCreator.RemoveLocalUsersFromParty(party, (from xu in party.Members where typeof(xu) == typeof(XboxUser)));

**Notes:**

* Attempting to retrieve party members or the unique party identifier of a party associated with a specific title ID before the call to **AddLocalUsersToParty** will throw **NoLocalSignedInUsersInParty** exception. This exception can also be interpreted as “party doesn’t exist”.
* **RemoveLocalUsersFromParty** is the only party manipulation API that will succeed even if the party doesn’t exist.
* Remote users cannot be removed from the party explicitly. Indirectly, however, remote users are removed when all local users are deleted from the party - the party is disbanded and no longer exists.

## Inviting remote users to a party

Users on other consoles (represented by the **XboxRemoteUser** class) can be invited to join a party using **XboxUser**’s **InviteToParty** method:

// Invite remote users to the party

partyCreator.InviteToParty(party, (from xu in anotherConsole.Users select XboxRemoteUser.FromXuid(this.Console, xu.Xuid)));

Note that the XUID’s of remote users can be plugged in directly as strings, or retrieved using the object model after connecting to the console they are local to (as shown above).

Remote users are invited to the “local” party from the perspective of “local” users on the console that created the party (console A). From the perspective of users that were invited (which are local to a console B), the party created on console A is the remote party.

**Notes:**

* **XboxUserNotSignedInException** will be thrown if the user object on which this function is called is not signed in.
* **InvalidXuidException** will be thrown if the user object cannot invite remote party members, or if invalid remote party members were specified.
* Inviting to a party that has no members will throw the **NoLocalSignedInUsersInParty** exception.

## Joining a remote party

Once a party invitation has been sent, an invited user on a console that did not create the party can either accept or decline the invitation using the **XboxRemoteParty** object.

invitedUser.AcceptInviteToParty(XboxRemoteParty.FromPartyId(remoteConsole, partyId));

invitedUser.DeclineInviteToParty(XboxRemoteParty.FromPartyId(remoteConsole, partyId));

**Note**:

* Just like **XboxRemoteUser**, the **XboxRemoteParty** object represents a party created on different console to the one accepting/rejecting the invitation, using a unique party identifier. The party identifier has to be retrieved for the console that created the party and passed in to the code that accepts or declines the party invitations for invited users.
* **InvalidXuidException** or **InvalidPartyInviteException** will be thrown if the user cannot accept or decline the party invitation because of not being invited or another Live service-specific reason. More information on conditions for either exception to be thrown is not available at this time.

## The lifetime of a party

This section will summarize the process of creating a party, inviting users, accepting invitations, and removing users from party using two consoles.

|  |  |
| --- | --- |
| **Console A** | **Console B** |
| User local to *Console A* **creates** a party for a specific Title ID and adds 2 other local users.   * A party is created with 3 users, the party creator and 2 other users specified. * The party is associated with a specific title. The same users could be in a different party for a different title, but only one party can exist for each title. * A unique party ID is assigned to the party. * Party members can be retrieved.   partyCreator.AddLocalUsersToParty(  XboxParty.FromTitleId(  consoleA,  TestTitleId),  addUsers  ); |  |
| One of the local party members **invite** several users from *Console B* to the newly created party, using its unique party ID.  partyCreator.InviteToParty(  party,  (from  xu in consoleB.Users  select  XboxRemoteUser.FromXuid(  consoleA,  xu.Xuid  )  )); |  |
|  | The users on *Console B* who were invited to the party for the same Title ID can either **accept** or **decline** the invitation.  firstInvitedUser.AcceptInviteToParty(  XboxRemoteParty.FromPartyId(  consoleB,  partyId  )  );  secondInvitedUser.DeclineInviteToParty(  XboxRemoteParty.FromPartyId(  consoleB,  partyId  )  ); |
| Users on *Console B* that accepted the invitation appear on the party Members list, and users that did not accept never become party members.  XboxUserBase[] members =  party.Members.ToArray(); |  |
| Party can be disbanded by signing out all local users that belong to it on the console that created the party (*Console A*). Alternatively, RemoveLocalUsersFromParty can be called, passing the party creator and all local party members:  partyCreator.RemoveLocalUsersFromParty(  party,  previouslyAddedUsers  ); |  |

# How to take a screenshot using XboxConsole

Beginning with the March 2014 release, the Xbox One XDK includes functionality that enables users to capture uncompressed images of the console’s frame buffer. The XboxConsole library exposes this through XboxConsole.CaptureScreenshot method. This method returns a BitmapSource object containing the captured frame buffer, which can then be used to render the image in a GUI or write the image to a file, among other tasks.

For example, to write the image to a file, do the following:

using (XboxConsole xbc = new XboxConsole())

{

BitmapSource screenshot = xbc.CaptureScreenshot();

JpegBitmapEncoder encoder = new JpegBitmapEncoder();

// Set highest quality level to avoid lossy compression artefacts

encoder.QualityLevel = 100;

encoder.Frames.Add(BitmapFrame.Create(screenshot));

using (FileStream file = new FileStream(@"C:\Screenshots\sample.jpg", FileMode.CreateNew))

{

encoder.Save(file);

}

}

# How to record a DVR capture using XboxConsole

As of the August 2014 release, the Xbox One XDK enables users to leverage GameDVR service on the console to capture in-game videos (MP4 format) between 6 and 300 seconds. The XboxConsole library exposes this through the XboxConsole.CaptureRecordedGameClip method. This method takes the full path of the file to write the recorded MP4 file to and the number of seconds of video to return from the “current time” backwards. For example, to capture a 30-second clip:

using (XboxConsole xbc = new XboxConsole())

{

xbc.CaptureRecordedGameClip(@"C:\DVRCapture\Clip.mp4", 30);

}

# How to push deploy an application to the Xbox One Console

Deployment to the Xbox One console can be done manually through the Visual Studio **Deploy Solution** option. However, if you would like to automate the deployment process, you can use XboxConsole to do push deployment to the console. Push deployment sends the content of the “\loose” folder (i.e.: files and folders within it) of an application on the PC to the console.

To deploy an application using push deployment, do the following:

using (XboxConsole xbc = new XboxConsole())

{

XboxPackage p = await xbc.DeployPushAsync(@"c:\temp\testapplication\loose", true);

}

The first argument to the DeployPushAsync takes a path to the “\loose” folder. The entire content of that folder will be deployed to the console. The second argument specifies whether extra files not present in deployFilePath should be removed – true means extra files will be removed and false means extra files are not removed.

The DeployPushAsync method is asynchronous, utilizing the [TAP pattern](http://msdn.microsoft.com/en-us/library/hh873175(v=vs.110).aspx). XboxConsole provides 4 overloads for this method:

1. Normal:

Task<XboxPackage> DeployPushAsync(string deployFilePath, bool removeExtraFiles)

1. With Progress Reporting:

Task<XboxPackage> DeployPushAsync(

string deployFilePath,

bool removeExtraFiles,

IProgress<XboxDeploymentMetric> progressMetric,

IProgress<XboxDeploymentError> progressError,

IProgress<XboxDeploymentExtraFile> progressExtraFile)

1. With Cancellation:

Task<XboxPackage> DeployPushAsync(

string deployFilePath,

bool removeExtraFiles,

CancellationToken cancellationToken)

1. With Progress Reporting and Cancellation:

Task<XboxPackage> DeployPushAsync(

string deployFilePath,

bool removeExtraFiles,

CancellationToken cancellationToken,

IProgress<XboxDeploymentMetric> progressMetric,

IProgress<XboxDeploymentError> progressError,

IProgress<XboxDeploymentExtraFile> progressExtraFile)

### Progress Reporting

DeployPushAsync overloads that support **progress reporting** accept several System.IProgress<T> arguments to report the progress of the deployment. Pass null for any of these arguments when that type of reporting is not required:

* The IProgress<XboxDeploymentMetric> progressMetric argument provides progress reports about various metrics of the transfer, e.g. the percentage of bytes that have been transferred.
* The IProgress<XboxDeploymentError> progressError argument provides reports about errors.
* The IProgress<XboxDeploymentExtraFile> progressExtraFile argument provides reports about any extra files of the same that might already exist in the console but are not in the deployment.

The following is an example of how to use these progress arguments.

// Using XboxDeploymentMetric from handler that gets called on progress updates

Progress<XboxDeploymentMetric> metric = new Progress<XboxDeploymentMetric>(

m => Console.WriteLine(m.TotalBytesTransferred + " bytes transferred."));

// Using XboxDeploymentError from handler that gets called on error

Progress<XboxDeploymentError> error = new Progress<XboxDeploymentError>(

e => Console.WriteLine("There was an error during deployment: " + e.ErrorCode));

// Using XboxDeploymentExtraFile from handler for extra files found in destination

Progress<XboxDeploymentExtraFile> extraFile = new Progress<XboxDeploymentExtraFile>(

f => Console.WriteLine("An extra file was found: " + f.FilePath));

// Deploying with all progress objects enabled

XboxPackage package = await xbc.DeployPushAsync(@"c:\temp\testpackage\", true, metric, error, extraFile);

If you only wanted the metric’s progress, you could do the following:

XboxPackage package = await xbc.DeployPushAsync(@"c:\temp\testpackage\", true, metric, null, null);

**Notes**:

* To use the progress arguments, make sure you include a using statement in your project to Microsoft.Internal.GamesTest.Xbox.Deployment.
* You can deploy on top of an application that is already deployed to the console. In this case, the progress arguments would report on changes to the existing deployment.

### Deployment Cancellation

DeployPushAsync overloads that support **cancellation** accept a CancellationToken. This allows the user to request the asynchronous operation to be cancelled by calling Cancel() on the token source (also see [MSDN example](http://msdn.microsoft.com/en-us/library/dd997364(v=vs.110).aspx)):

CancellationTokenSource tokenSource = new CancellationTokenSource();

Task<XboxPackage> deployTask = xbc.DeployPushAsync(

@"c:\temp\testapplication\loose", true, tokenSource.Token);

tokenSource.Cancel();

As described in the MSDN article linked above, waiting on a task when it transitions to Canceled state will throw a TaskCanceledException.Task exception. If the calling code does not wait on the task, the task status will simply be set to Canceled and [continuations](http://msdn.microsoft.com/en-us/library/ee372288(v=vs.110).aspx) will be executed unless the task continuation options specify as NotOnCanceled.

# How to register a package located on TitleScratch drive

To facilitate faster switching between different versions of the same application on Xbox One, keep loose files for each package in folders on the **TitleScratch** drive then register/unregister each one as necessary. A registered **TitleScratch** package appears in the applications list (or in the collection returned by **InstalledPackages** property of XboxConsole) like any other application. **Uninstalling** such a package will remove the associated files on the **TitleScratch** drive, which is usually not desired. Instead, **Unregister** the package to remove it from the applications list while keeping the files on the drive.

Use the **Register** method of XboxConsole to register a package:

using (XboxConsole xbc = new XboxConsole())

{

XboxPackage package = xbc.RegisterPackage(

new XboxPath(@"\mygame", XboxOperatingSystem.Title));

}

This method takes only one parameter, an **XboxPath** containing a relative path to the loose package files on the **TitleScratch** drive. This doesn’t include the root and simply starts with a backslash.

To **Unregister** a package that has been previously registered (not push-deployed), use the **Unregister** method of the package class:

package.Unregister();

Attempting to unregister a package deployed with push-deployment will result in an **InvalidOperationException**.

# How to get available space for application installation

To get the space available for package installation use the **GetAvailableSpaceForAppInstallation** method of **XboxConsole**:

using (XboxConsole xbc = new XboxConsole())

{

ulong spaceAvailable = xbc.GetAvailableSpaceForAppInstallation();

}

The available space is returned in bytes, as a 64-bit unsigned value. An example of calculating how much space is required to install a package (for making a comparison to determine whether there is enough space) is provided in **XtfGetAvailableSpaceForAppInstallation** topic in XDK documentation.

# How to control the lifecycle of a package on the Xbox One console

A package contains a collection of applications and other resources; an *app package* includes a manifest that lists all package components and executable applications. Using **XboxConsole**, you can access your package and issue commands to it.

While launching is done on a per application basis, all other states are controlled by the package; that is, packages are responsible for suspending, resuming, constraining, constraining, and terminating themselves. To control your package, get a reference to it by querying the **InstalledPackages** member of an **XboxConsole** object. You can iterate through the **InstalledPackages** enumerable or look it up by name or ID. In each **XboxPackage** is an enumerable of **XboxApplication** objects that contains a list of all of the associated applications belonging to that package. You can iterate through it or look it up by **ApplicationID** to locate the specific application.

This code example shows how to quickly look up an application from a package using the Linq syntax using the package’s **FamilyName,** and the application’s **ApplicationID**. You could use the package’s **FullName** or the application’s **Aumid** (Application User Model ID). Once you have a reference to your package and application, you can control the lifecycle accordingly.

using (XboxConsole xbc = new XboxConsole())

{

XboxPackage p = xbc.InstalledPackages.First(package => package.FamilyName.Contains("XboxConsole.XboxSample"));

XboxApplication a = p.First(app => app.ApplicationID == "App");

a.Launch();

p.Suspend();

p.Resume();

p.Constrain();

p.Unconstrain();

p.Terminate();

}

To launch an application with command line arguments, use the overloaded Launch() method with *arguments* parameter:

a.Launch(commandLineArguments);

These will be passed into target application’s OnActivated handler (consult platform documentation for details).

### Separation of functionality

The division of functionality between the **XboxApplication** and **XboxPackage** objects are explicitly listed in the following table.

|  |  |
| --- | --- |
| XboxPackage | XboxApplication |
| * Suspend() * Resume() * Constrain() * Unconstrain() * Terminate() * Unsnap() * Uninstall() | * Launch() * Launch(arguments) * Snap() |

### Registering an event handler for the ExecutionStateChanged event

On the other hand, if you only wanted to take action on a package whenever there is a state change, then you can register an event handler for the **ExecutionStateChanged** event. This event relies on the results of an asynchronous polling thread, which checks the **ExecutionState** of the associated package once every second. Once you’ve registered for the event, whenever the package’s state changes your tool can *“Do Something”* about it.

using (XboxConsole xbc = new XboxConsole())

{

XboxPackage p = xbc.InstalledPackages.First(package => package.FamilyName.Contains("XboxConsole.XboxSample"));

p.ExecutionStateChanged += (sender, e) =>

{

switch (e.NewState)

{

case PackageExecutionState.Constrained: /\* Do Something \*/ break;

case PackageExecutionState.Running: /\* Do Something \*/ break;

case PackageExecutionState.Suspending: /\* Do Something \*/ break;

case PackageExecutionState.Suspended: /\* Do Something \*/ break;

case PackageExecutionState.Terminated: /\* Do Something \*/ break;

case PackageExecutionState.Unknown: /\* Do Something \*/ break;

}

};

}

**Notes:**

* The **FamilyName** string consists of two different pieces of information concatenated together with underscores. These two information are: **PckageName\_PublisherId**.
* The **FullName** string consist of various pieces of information concatenated together with underscores. These pieces of information are: **PackageName\_VersionNumber\_ProcessorArchitecture\_ResourceId\_PublisherId**. All pieces of this string are required accept for **ResourceId**, which can be empty.
* The **Aumid** string consists of the **FamilyName** and **ApplicationId** concatenated by an “!”. These two information are: **FamilyName!Applicationid**.
* Only SRA applications are able to be snapped. In the event that the user calls **Snap** on an application that cannot be snapped, an **XboxSnapException** will be thrown.
* At any time during the lifecycle of your package, you can check what state it is in by calling the **ExecutionState** property.
* The **XboxPackage** is in a **Running** state only if any of its applications are running.
* The **Suspending** and **Terminated** state of an **XboxPackage** only exist for a brief period, so checking for those is not always reliable.
  + Larger applications will remain in the **Suspending** state for several seconds, but in general it is more reliable to check for the **Suspended** state.
  + Once an application has been closed, after it leaves the **Terminated** state, it enters the **Unknown** state, so in general these two states can be treated similarly.
* As a reminder, when a package on the console is launched, suspended, or resumed it does not necessarily enter immediately into the expected state.
  + Packages being launched enter the **Running** state, then the **Constrained** state, and then the **Running** state once again.
  + Packages being suspended enter the **Constrained** state before entering the **Suspended** state.Packages being resumed return to the **Constrained** state.

# How to uninstall a package

The **XboxPackage.Uninstall** method (accepting no arguments) can be used to uninstall a package once its reference has been found in **InstalledPackages** collection of **XboxConsole** object.

using (XboxConsole xbc = new XboxConsole())

{

XboxPackage p = xbc.InstalledPackages.First(package => package.FamilyName.Contains("XboxConsole.XboxSample"));

p.Uninstall();

}

# How to launch a non-package executable

On Xbox One development kits, users are able to launch more than just packaged applications. With the **XboxProcess.Run** static method, the **XboxConsole** library exposes functionality that enables users to launch any viable executable that exists on the console’s hard drive, in either operating system. The method takes as its parameters an **XboxConsole** object, the full path of the executable to be launched, the arguments to be supplied to the executable at launch, and an **XboxOperatingSystem** enumvalue.

using(XboxConsole xbc = new XboxConsole())

{

XboxProcess.Run(

xbc,

@"C:\windows\system32\cmd",

"/C echo Hello World",

XboxOperatingSystem.System);

}

Note that the path to the executable is formatted in accordance with Windows PC standards—that is, it is *not* formatted like Xbox One package paths, which are prefaced with an ‘X.’

The **XboxProcess.Run** method comes with an additional overload that allows users to capture the redirected standard output of the launched application. This overload accepts as its final parameter an **Action<string>** delegate, where the string parameter corresponds to the output received from the running executable.

using(XboxConsole xbc = new XboxConsole())

{

XboxProcess.Run(

xbc,

@"C:\windows\system32\cmd",

"/C echo Hello World",

XboxOperatingSystem.System,

(outputReceived) => {/\* Do something with the output \*/});

}

It is important to note that lifecycle management of the launched executable will be left entirely up to the user. This means that any executable that does not self-terminate will have to be manually terminated. Fortunately, this can be easily accomplished via the Windows **kill** command.

using (XboxConsole xbc = new XboxConsole())

{

XboxProcess.Run(

xbc,

@"C:\windows\system32\cmd",

string.Empty,

XboxOperatingSystem.System);

XboxProcess.Run(

xbc,

@"C:\windows\system32\kill",

"cmd",

XboxOperatingSystem.System);

}

# How to copy files and directories

The **XboxConsole** library mirrors IO operations to that of the .NET library’s **FileSystemIO**. In the context of the Xbox One console, you can **Copy** files and directories from your PC to the console and you can also **Copy** files and directories from the console to your PC. For more information on what other file IO operations are available, see the references for **XboxFile**, **XboxFileInfo**, **XboxDirectory** and **XboxDirectoryInfo** classes.

## Copying files to and from the System OS

The Xbox One runs three operating systems. When your title is installed, the Xbox One console stores your package in the SRA environment. When this package is launched, the Xbox One console spins up another OS and runs your package in an ERA environment. When you want to access files and directories on the Xbox One console, you must specify which OS environment you want to access via the XboxPath object. The XboxPath takes an argument of type XboxOperatingSystem. This enumerable specifies which operating system the path should map to. For instance the XboxOperatingSystem.System would map to the SRA system while the XboxOperatingSystem.Title would map to the ERA system. This section focuses on copying files within the System OS. The next section will cover the differences when accessing the Title OS.

The **XboxConsole** library provides an **XboxFile** class with only static member methods while it also offers an **XboxFileInfo** class that you can instantiate. Depending on your need, you may want to just use a method and not care about keeping track of the instance then you can use the static methods from the **XboxFile** class, otherwise, use the **XboxFileInfo** class. While both **Copy** methods offer similar functionalities, their method signatures are slightly different.

For example, to copy a file from the PC to the console, the static **Copy** signature looks like this:

public static void Copy(string sourceFile, XboxPath destinationFile, XboxConsole console)

To copy a file from the console to your PC, the static **Copy** signature looks like this:

public static void Copy(XboxPath sourceFile, string destinationFile, XboxConsole console)

In the static **Copy** method, the method is overloaded where the direction of the source to destination is determined by the order of the parameter’s type. So, to call this static method in your test tool, simply give it the proper parameters. For example, to copy a file from your PC to the console, call the static method like this:

using (XboxConsole xbc = new XboxConsole())

{

//Copy from PC to console

XboxFile.Copy(@"c:\temp\testfile.txt", new XboxPath(@"xd:\testdir\testfile.txt", XboxOperatingSystem.System), xbc);

}

On the other hand, if you want to use instance objects to copy files then you will need to use two objects - depending on the direction of the copy.

1. If you want to copy files from your PC to the console, create a **System.IO.FileInfo** object. This is because a **FileInfo** object is an object that represents a file on your PC. It does not know about your console’s file structures. To use this object, you will need to provide a path on the console and a reference to the console object.

To use the **FileInfo** object to copy files from the PC to the console, do this:

//Copy from PC to console

FileInfo fileInfo = new FileInfo(@"c:\temp\testfile.txt");

fileInfo.CopyTo(new XboxPath(@"xd:\testdir\", XboxOperatingSystem.System), xbc);

1. If you want to copy files from the console to the PC, create an **XboxFileInfo** object. Using this object’s **Copy** method, all you need to do is give it a local path on your PC. This is because an **XboxFileInfo** object represents a file on an Xbox. It does not know about your PC file structures.

To use the **XboxFileInfo** to copy files from the console to the PC, do this:

//Copy from console to PC

XboxFileInfo xbfileInfo = new XboxFileInfo(new XboxPath(@"xd:\testdir\testfile.txt", XboxOperatingSystem.System), xbc);

xbfileInfo.Copy(@"c:\temp\xd\");

Once the operation completes successfully, you can check the respective objects to verify that the file exists on the console.

* For the **XboxFile** object, the static **Exists** method accepts an **XboxPath** object containing the path to the file on the console, it also takes a reference to the console. You can call the static **Exists** method as follows:

if(XboxFile.Exists(new XboxPath(@"xd:\testdir\testfile.txt", XboxOperatingSystem.System), xbc))

* For the **XboxFileInfo** instance, calling the **Exists** member will check if the file exists on the console. You can call the **Exists** member as follows:

if(xbfileInfo.Exists)

* For the **FileInfo** instance, calling the **Exists** member will check if a file exists on the PC. You can call the **Exists** member as follows:

if(fileInfo.Exists)

**Notes**:

* Make sure you include these using statements:
  + using Microsoft.Internal.GamesTest.Xbox;
  + using Microsoft.Internal.GamesTest.Xbox.IO;
* When defining a directory, be sure to include the trailing “\”. For example:

@"c:\temp\xd\"

@"xd:\testdir\"

* You can rename a file on the destination while making a **Copy**. Instead of providing just a destination folder, provide the full file path. For example:

//Copy from PC to console

XboxFile.Copy(@"c:\temp\testfile.txt", new XboxPath(@"xd:\testdir\mytestfile.txt", XboxOperatingSystem.System), xbc);

* Be sure to catch and handle exceptions gracefully; such as **FileNotFoundException,** **XboxConsoleException** and other common file IO exceptions. For more exceptions to catch, see **Remarks** section.
* One important point to keep in mind when working with files and directories using instance objects is that the **XboxFileInfo** and **XboxDirectoryInfo** objects are instances that represents files or directories on the console. To do IO on your PC, you can instantiate **System.IO.FileInfo** and **System.IO.DirectoryInfo** objects instead.
* The XboxPath takes an argument of type XboxOperatingSystem. This enumeration specifies which operating system the path should map to. For instance the XboxOperatingSystem.System would map to the SRA system while the XboxOperatingSystem.Title would map to the ERA system.
* To assist in verifying that your IO operations are successful, use the Xbox One XDK command prompt to inspect folder contents and verify that your files are copied over to the console as you expect. For a list of possible commands, run the command: **dir xb\***
* For more information on how Xbox One handles file copy operation, see the [Documentation](https://developer.xboxlive.com/en-us/platform/development/documentation/Pages/home.aspx) page. To get to the **Remote File Copy** topic, drill down the TOC path: **Xbox One XDK/DevTools and Automation/Tools/Remote Console**.

## Copying files to and from the Title OS

Copying files to and from the Title OS is exactly the same as copying to and from the System OS. The two primary differences are:

1. A package must be running for the Title OS to exist
2. The **XboxPath** now takes an XboxOperatingSystem.Title

So, to copy a file to and from the Title OS, do the following.

using (XboxConsole xbc = new XboxConsole())

{

XboxPackage p = xbc.InstalledPackages.First(p2 => p2.FamilyName.Contains("XboxConsole.XboxSample"));

p.Applications.First().Launch();

FileInfo fileInfo = new FileInfo(@"c:\temp\testfile.txt");

fileInfo.CopyTo(new XboxPath(@"xd:\testdir\", XboxOperatingSystem.Title), xbc); //Copy from PC to console

XboxFileInfo xbfileInfo = new XboxFileInfo(new XboxPath(@"xd:\testdir\testfile.txt", XboxOperatingSystem.Title), xbc);

xbfileInfo.Copy(@"c:\temp\xd\"); //Copy from console to PC

p.Terminate();

}

Notes:

* Even though the code above references the same “xd:\testdir\” as previous sample code; however, this drive exists on the Title OS and is completely different than the “xd:\testdir\” folder on the System OS.

## Copying files to and from an application container

The process for copying files relative to the context of a package is the same as copying files to and from drive partitions. The main difference is in how the path is constructed. While the process described above starts with a drive letter on the console, to get the relative path of a package, you can use this syntax:

{<Package.FullName>}:\

That is, the relative path to a package is, essentially, the package’s **FullName** wrapped inside the curly braces followed by “:\”. This means that you will need to have a reference to the package you want to make a copy to.

To make a copy to the console relative to the context of your package, do the following:

using (XboxConsole xbc = new XboxConsole())

{

XboxPackage p = xbc.InstalledPackages.First(p2 => p2.FamilyName.Contains("XboxConsole.XboxSample"));

XboxPath xboxDir = new XboxPath(string.Format(@"{0}{1}{2}:\testdir\", "{", p.FullName, "}"), XboxOperatingSystem.System);

XboxFile.Copy(@"c:\temp\testfile.txt", xboxDir, xbc); //Copy from PC to console

}

**Note**:

* Copying files to and from relative path of a package is possible when the package is not running. If you attempt to read or write to a relative path where the package is running, you will get an **XboxConsoleException** exception; with - **Reason: The requested resource is in use.** To gain access to this path again, you must terminate the package. You can, however, still read and write to other drive partitions that are not locked by the running process on the console.
* When the package is launched, however, copying files to and from this relative path is also not accessible under the Title OS. You will get an **XboxConsoleException** exception; with - **Reason: The media is write protected**.

## Copying files to and from the @scratch drive

While your package is running, it may have written files to a scratch drive. For example your title may write crash dump files to this scratch location. Normally, to get the files out of this location, you will need to launch the package to gain access to this drive (e.g.: xd:\testfile.dmp). With the **{@scratch}:\** syntax, you can access this scratch location when your package is not running.

To copy file from this scratch location to your PC, do this:

//Copy from scratch drive to PC

XboxFile.Copy(new XboxPath(@"{@scratch}:\testfile.dmp", XboxOperatingSystem.System), @"c:\temp\testfile.txt", xbc);

## Copying directories to and from the Xbox One console

The process for copying a directory is the same as copying a file to and from the console. The only difference is that you will be using the **XboxDirectory** and **XboxDirectoryInfo** classes instead. Accessing the System OS, Title OS, and @scratch drive is also the same.

Here is an example of how to copy a directory from your PC to the console.

//Copy directory from PC to console

XboxDirectory.Copy(@"c:\temp", new XboxPath(@"xd:\testdir", XboxOperatingSystem.System), xbc);

This call will make a copy of everything that is inside of “c:\temp”, including subdirectories, to the “xd:\testdir” on the console. To copy a directory from the console to the PC, just switch the source and destination arguments.

If you want to use an instance of the **XboxDirectoryInfo**, you can use the **Copy** and **CopyTo** method to do the same work. There is a *recursive* flag that might be of interest to you. This flag allows you to choose whether you want to recursively copy all the subdirectories.

Here is an example of how to copy a directory, excluding subdirectories, from your console to the PC.

//Copy a directory from console to PC, excluding subdirectories

XboxDirectoryInfo xdi = new XboxDirectoryInfo(new XboxPath(@"xd:\testdir", XboxOperatingSystem.System), xbc);

xdi.Copy(@"c:\temp", false);

In the sample code above, the **Copy** method takes an additional Boolean parameter as a *recursive* flag. If this flag is true, this method will copy the directory and all subdirectories. If false, this method only copy the directory, excluding subdirectories.

## Getting progress updates while copying

Each **Copy** method, provided by the **XboxConsole** library, has an overload that takes an **IProgress<XboxFileTransferMetric>** object. These overloads can be used to track the progress of a copy. For Example:

XboxFile.Copy(

@"c:\temp\testfile.txt",

new XboxPath(@"xd:\testdir\testfile.txt", XboxOperatingSystem.System),

xbc,

new Progress<XboxFileTransferMetric>(metrics =>

{

Console.WriteLine("File Size: " + metrics.FileSizeInBytes +

" Transferred: " + metrics.FileBytesTransferred);

}));

Progress updates with directories work the same way:

XboxDirectory.Copy(

@"c:\temp\",

new XboxPath(@"xd:\testdir\", XboxOperatingSystem.System),

xbc,

new Progress<XboxFileTransferMetric>(metrics =>

{

Console.WriteLine("File being copied: " + metrics.SourceFilePath +

" Total size of transfer: " + metrics.TotalSizeInBytes +

" File Size: " + metrics.FileSizeInBytes +

" Transferred: " + metrics.FileBytesTransferred);

}));

The **XboxFileTransferMetric** class provides information about how far along a copying operation for both individual files and directories. For more information about what is provided see the **XboxFileTransferMetric** class references.

**Notes:**

* Exceptions **must** **not** be thrown in the progress handling method. Throwing an exception will result in undefined behavior.
* When copying directories, multiple files may be transferred at the same time. As such, progress reports about different files may also be called at the same time or slightly out of order. This can lead to slight discrepancies in reports about the total bytes that have been transferred. For example, a report about a file A.txt could say that the total amount of bytes that have been transferred is 10000 bytes, and the next report that comes in about a file B.txt saying that the total amount transferred so far is 9500 bytes. However, the order of reports for a given file will always be correct.

# How to capture debug output from the Xbox One console

To capture debug output from your title running on the console, you need do two things:

1. Make sure your title outputs debug messages using OutputDebugString.
2. Create an event handler on your client-side test tool and attach it to your title’s running process.

Assuming your title already outputs messages using OutputDebugString, you can attach event handlers to the running process like this.

XboxConsole xbc = new XboxConsole();

XboxPackage p = xbc.InstalledPackages.First(p2 => p2.FamilyName.Contains("XboxSample"));

p.Applications.First().Launch();

//Defining the handler

bool isHandlerCalled = false;

EventHandler<TextEventArgs> handler = (sender, args) =>

{

isHandlerCalled = true;

};

XboxProcess process = xbc.GetRunningProcesses(XboxOperatingSystem.Title).Single(p => p.ImageFileName.Contains("XboxConsole.XboxSample"));

//Attaching our event handler to the process

process.TextReceived += handler;

...

xbc.Dispose();

When you no longer want to capture debug outputs, simply detach your hander from the process. Like this.

//Detaching our event handler from the process

process.TextReceived -= handler;

**Notes**:

* Make sure your package is running before attempting to attach or detach your handler.
* Note that a running package is run under the XboxOperatingSystem.Title environment.
* Do not dispose your XboxConsole object while you still want to receive debug output. A disposed object cannot continue to receive debug output.

# How to simulate controller inputs

Using **XboxConsole**, you can simulate controller inputs. Instead of having to physically press a button on the controller, you can programmatically send a gamepad state to the console.

The following code example shows how to connect the gamepad to the console and send it a list of buttons that the user has pressed.

using (XboxConsole xbc = new XboxConsole())

{

var gamepad = xbc.CreateXboxGamepad();

gamepad.Connect();

Thread.Sleep(500);

gamepad.SetXboxGamepadState(new XboxGamepadState()

{

Buttons = XboxGamepadButtons.DpadDown | XboxGamepadButtons.DpadRight,

RightTrigger = 1.0f,

LeftTrigger = 0.01f,

LeftThumbstickX = -1.0f,

RightThumbstickY = 1.0f

});

Thread.Sleep(500);

gamepad.Disconnect();

}

In the example above, we created the **XboxGamepad** via the **XboxConsole** object. We then connect the **XboxGamepad** to the console. After the virtual **XboxGamepad** is connected, we set the state for the Down and Right D-pad, the Right Trigger, the Left Thumb Stick, and the Right Thumb Stick. This represents a user pressing down on those buttons at the same time.

To simulate a user has released a button, you will need to send another **SetXboxGamepadState** by excluding the buttons that’s being released or for the inputs with floating points, set them to zero. In other words, just send the buttons that are currently still being pressed. For example, if we want to simulate that the user has released the Down D-pad and the Left Thumb Stick, we will issue another **SetXboxGamepadState** as follows:

gamepad.SetXboxGamepadState(new XboxGamepadState()

{

Buttons = XboxGamepadButtons.DpadRight,

RightTrigger = 1.0f,

LeftTrigger = 0.01f,

LeftThumbstickX = 0.0f,

RightThumbstickY = 1.0f

});

Noticed that the **DpadDown** is not in the list of **Buttons** and the **LeftThumbstickX** is set to 0.0f. To simulate that the user has released all the buttons, send this state:

gamepad.SetXboxGamepadState(new XboxGamepadState());

In this case, all the previously pressed buttons are now released.

**Notes**:

* You can create as many **XboxGamepad** objects as you want; however, you can have no more than 16 gamepads connected at one time (inclusive of virtual and physical controllers).
* It is a good idea to wait after connecting or sending **XboxGamepad** states to the console. This gives the console a chance to register the call. This also means that if you call to set an **XboxGamepad** state and immediately query if the state has changed, it might not. Waiting a few milliseconds can help with that.
* You cannot set more than one **XboxGamepad** state at a time. That is, if you try to set more than one XboxGamepad state within a time period (roughly 25 ms), an exception will occur. If you are simulating multiple controllers at the same time, perhaps across threads or within tasks, consider using lock and sleep around calls to **SetXboxGamepadState**.Like this:  
  lock (staticLockObject)  
  {  
   gamepad.SetXboxGamepadState(state);  
   Thread.Sleep(25);  
  }
* **XboxGamepad** objects can be paired with a user in the game. See [How to pair a controller with a user](#_How_to_pair) section.
* Always disconnect an **XboxGamepad** after you are done so that new **XboxGamepad** objects can connect to it. If you cannot connect an **XboxGamepad** because all 16 slots are occupied, reboot the console and try connecting again.
* **XboxGamepadButtons** is an enumeration. For a complete list of possible input buttons, see the **XboxGamepadButtons** comments on the enumeration. For thumb sticks and triggers, they have range of values as follows:
  + Thumb sticks: between -1 to 1 as a float
  + Triggers: between 0 to 1 as float

# How to pair a controller with a user

**XboxConsole** can pair both virtual controllers and physical controllers with users. The process for doing this is fairly simple.

using (XboxConsole xbc = new XboxConsole(this.toolsIP))

{

XboxUser user = xbc.Users.First(x => x.EmailAddress == @"UserEmailAddress@xboxtest.com");

XboxGamepad gamepad = xbc.CreateXboxGamepad();

gamepad.Connect();

user.PairWithVirtualController(gamepad);

//Do something with the paired controller

gamepad.Disconnect();

}

In the example above, we pair a virtual controller with a user on the console that matches our desired email address. If you know the controller Id, you can pair a physical controller as well using the **PairWithPhysicalController** method of the **XboxUser** class.

Pairing a user with a virtual controller adds the controller to a list of controllers already associated with the user. Titles will enumerate this list to choose one controller for driving the gameplay and menu interaction. Physical controllers are assigned IDs in a lower range than virtual controllers so they are usually returned first during the enumeration process, and most titles simply grab the first item in the list. This causes any paired virtual controllers to be ignored as soon as a title launches, even though they may work on the Home screen.

To achieve **exclusive** pairing (pair a user with a virtual controller and discard any previous pairings), use the **PairWithVirtualControllerExclusive** method. This will ensure the user is paired with **only one** controller, the virtual controller that you specify. The process is the same as in the above code example:

user.PairWithVirtualControllerExclusive(gamepad);

**Notes:**

* The user has to be signed-in at the time of pairing.
* Pairing a controller automatically overrides the previous pairing that the controller had.

# Exception handling

The majority of exceptions generated by the **XboxConsole** library inherit from **XboxException**. Therefore, users are encourage to place all calls into the **XboxConsole** library inside of a try/catch block that will catch exceptions that derive from **XboxException**.

In addition to types that inherit from **XboxException**, the **XboxConsole** library will also throw some standard .NET exceptions depending on the operation being performed. For example, the I/O functions can throw typical I/O exceptions like **FileNotFoundException, PathTooLongException, UnauthorizedAccessException**, etc. Additionally, any **XboxConsole** method that takes a timeout parameter can also throw a **TimeoutException.**

# Remarks

* If you get an exception saying **XboxConsole** cannot find a default console on your PC then make sure you use **xbconnect** command to register a default console and try again.
* You must build your application as x64 because **XboxConsole** takes a dependence on the 64-bit binaries included in the XDK.
* If you receive an **XboxSignInException**, make sure you are using an email account from “xboxtest.com” domain. Only emails for accounts registered for the game sandbox are allowed. To obtain an email account for your use, contact your project manager.