GDK Extension

Welcome to the GDK Extension! In this document you can find the full API documentation and guides necessary to get you started with creating Microsoft Store games on Windows (64-bit only).

Guides

Please refer to these guides for building and setting up your project for use with the GDK Extension and configuring files related to the extension.

- **Building Guide** (how to build the extension from source)
- **Quick Start Guide** (setup project, sandboxes, configuration files)

API

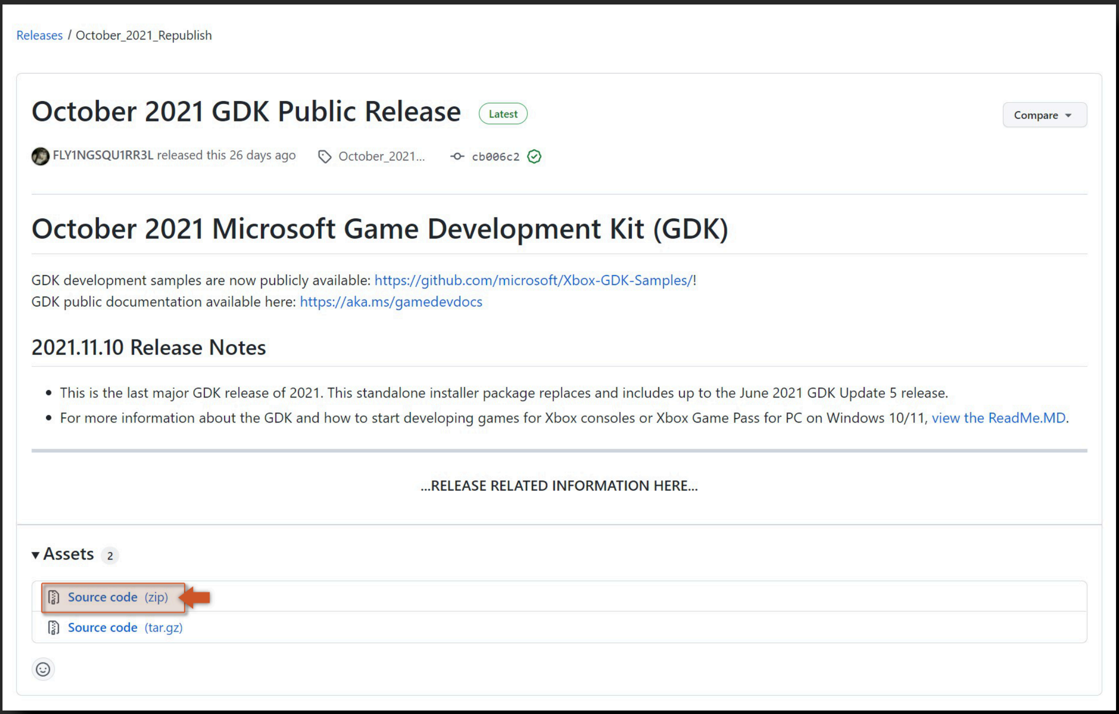
The GDK Extension API section includes documentation on all provided functions, which are divided into separate modules. There are a set of management functions that are required for the GDK Extension to be functional, and then the various GDK Extension modules (such as Storage, Xbox Live, In-App Purchases) can be used as you wish.

Please see GDK Extension API for details on the provided modules.

Building Guide

These guide will provide you the necessary steps for building the extension from source, if you intend to do so.

1. Install Visual Studio 2019 (see **download site**)
2. Install GDK (Oct 2021) from Microsoft GitHub repository (see **download site**)



Releases / October_2021_Republish

October 2021 GDK Public Release Latest

FLYINGSQU1RR3L released this 26 days ago · October_2021... · cb006c2

October 2021 Microsoft Game Development Kit (GDK)

GDK development samples are now publicly available: https://github.com/microsoft/Xbox-GDK-Samples/!
GDK public documentation available here: https://aka.ms/gamedevdocs

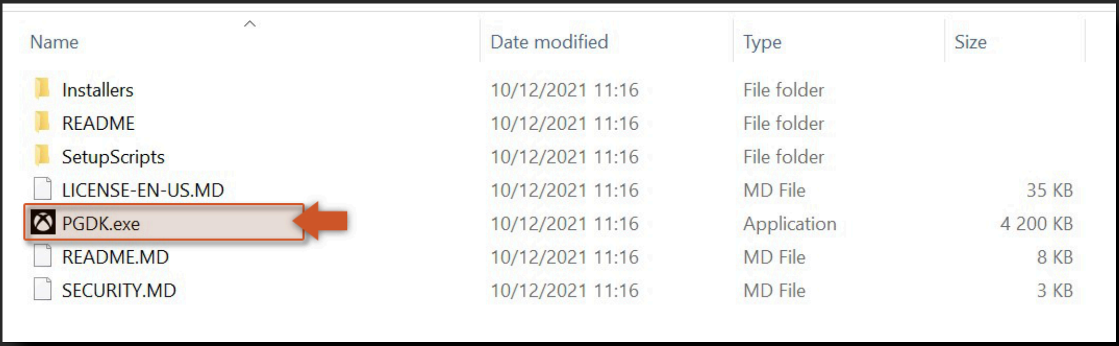
2021.11.10 Release Notes

- This is the last major GDK release of 2021. This standalone installer package replaces and includes up to the June 2021 GDK Update 5 release.
- For more information about the GDK and how to start developing games for Xbox consoles or Xbox Game Pass for PC on Windows 10/11, view the ReadMe.MD.

...RELEASE RELATED INFORMATION HERE...

Assets 2

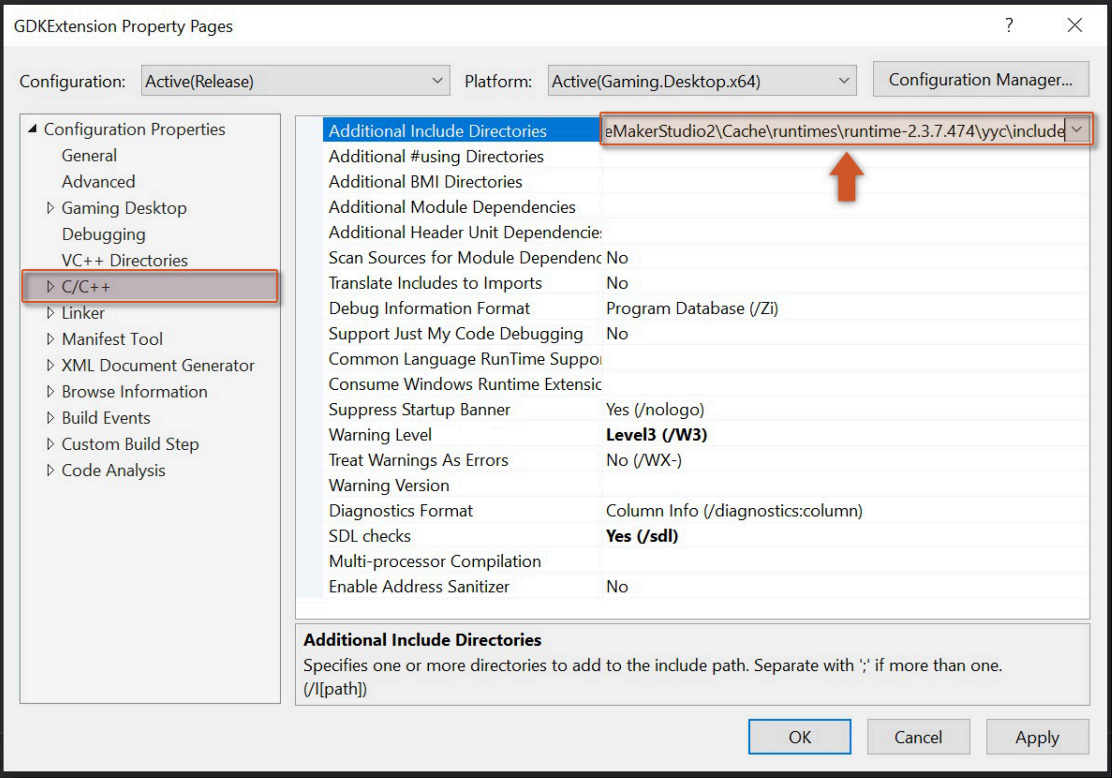
- Source code (zip)
- Source code (tar.gz)



Name	Date modified	Type	Size
Installers	10/12/2021 11:16	File folder	
README	10/12/2021 11:16	File folder	
SetupScripts	10/12/2021 11:16	File folder	
LICENSE-EN-US.MD	10/12/2021 11:16	MD File	35 KB
PGDK.exe	10/12/2021 11:16	Application	4 200 KB
README.MD	10/12/2021 11:16	MD File	8 KB
SECURITY.MD	10/12/2021 11:16	MD File	3 KB

3. Install CMAKE (see **download site**)
4. Clone the GDK Extension repository, has sub-modules (see **github repository**)
5. Open the Solution in `DLL/GDKExtension.sln` with Visual Studio 2019
6. In the Visual Studio IDE, go to `Project Properties` → **C/C++** → `Additional Include Directories` and add the following path:

   → `C:\ProgramData\GameMakerStudio2\Cache\runtimes\<current-runtime>\yyc\include\` (where `<current-runtime>` refers to the current runtime).

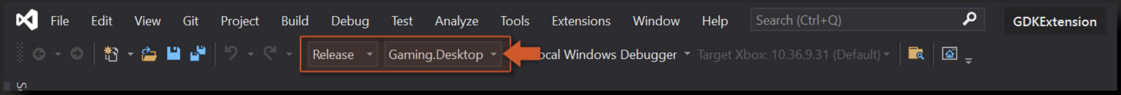

GDKExtension Property Pages

Configuration: Active(Release)   Platform: Active(Gaming.Desktop.x64)   Configuration Manager...

Additional Include Directories	eMakerStudio2\Cache\runtimes\runtime-2.3.7.474\yyc\include
Additional #using Directories	
Additional BMI Directories	
Additional Module Dependencies	
Additional Header Unit Dependencie	
Scan Sources for Module Dependenc	No
Translate Includes to Imports	No
Debug Information Format	Program Database (/Zi)
Support Just My Code Debugging	No
Common Language RunTime Suppo	
Consume Windows Runtime Extensic	
Suppress Startup Banner	Yes (/nologo)
Warning Level	**Level3 (/W3)**
Treat Warnings As Errors	No (/WX-)
Warning Version	
Diagnostics Format	Column Info (/diagnostics:column)
SDL checks	**Yes (/sdl)**
Multi-processor Compilation	
Enable Address Sanitizer	No

Additional Include Directories
Specifies one or more directories to add to the include path. Separate with ';' if more than one. (/I[path])

7. Select one of the building targets `Debug|Gaming.Desktop.x64` or `Release|Gaming.Desktop.x64` (other options won't work)



8. Build the project (the compiled DLL will be exported to the folder `GDK_Project_GMS2/extensions/GDKExtension/`.

9. Done, you've now finished the building process.

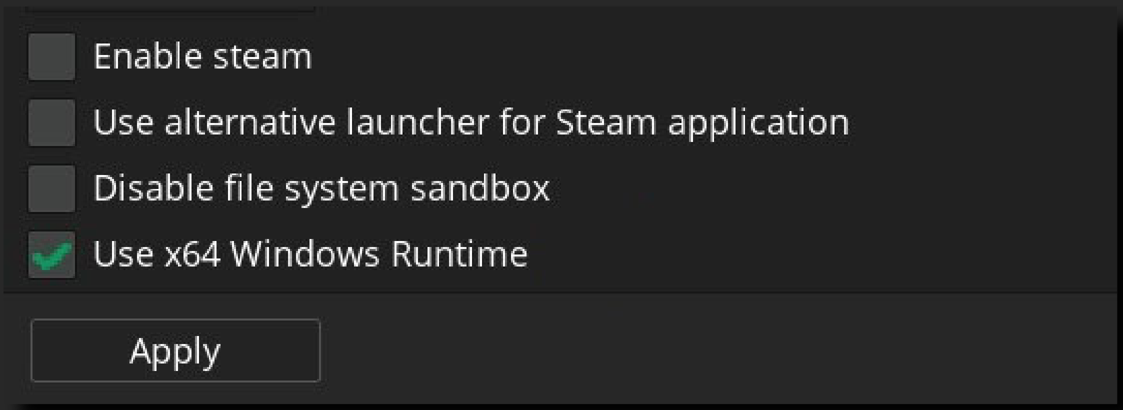
Quick Start Guide

You will initially need to set up your project for the GDK Extension to work properly. This guide covers all the aspects of your project that you will need to configure.

Project Setup

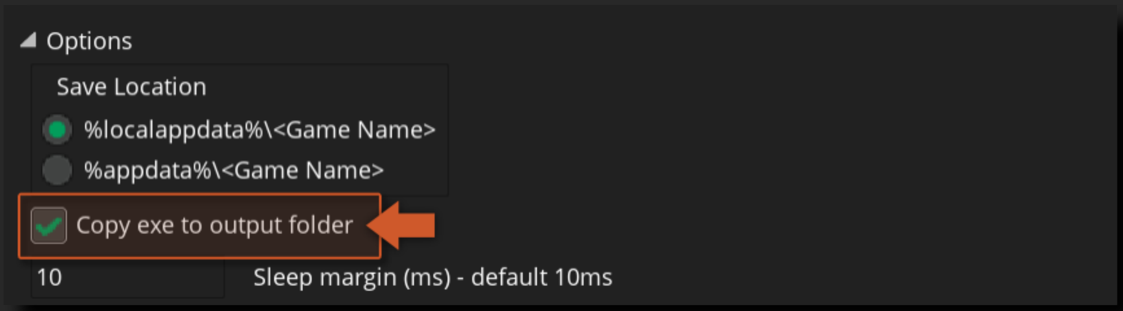
In order to build or run your project using the extension you need to import the extension in and change and make some changes to your project:

1. Import the local package provided with the public release version.
2. Use the 64-bit Windows runtime. To change this go into **Game Options** → **Windows** → **General** and enable "Use x64 Windows Runtime".



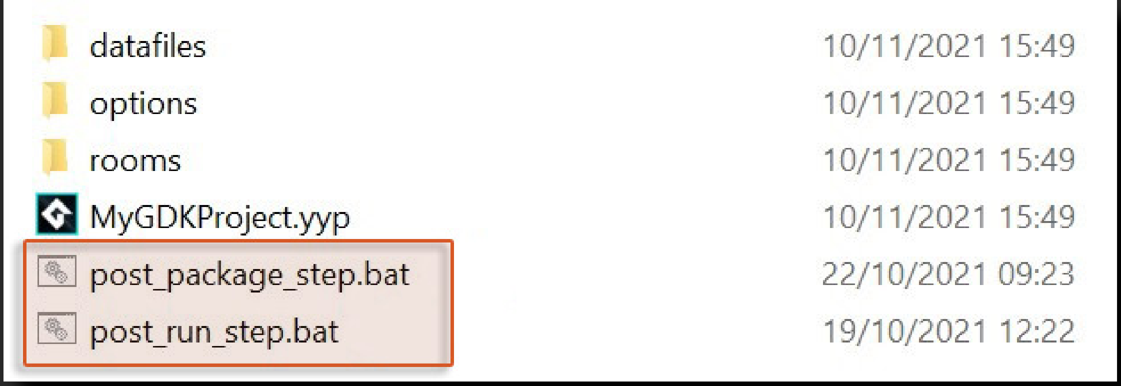
☐ Enable steam
☐ Use alternative launcher for Steam application
☐ Disable file system sandbox
✓ Use x64 Windows Runtime

3. Also make sure you enable the **"Copy exe to output folder"** setting:



Options
Save Location
● %localappdata%\<Game Name>
○ %appdata%\<Game Name>
✓ Copy exe to output folder
10 Sleep margin (ms) - default 10ms

4. The extension comes with two **.bat** files: `post_package_step.bat` and `post_run_step.bat`. These files need to be placed into your project's root folder, which is the same directory as your project's .yyp file.



datafiles	10/11/2021 15:49
options	10/11/2021 15:49
rooms	10/11/2021 15:49
MyGDKProject.yyp	10/11/2021 15:49
post_package_step.bat	22/10/2021 09:23
post_run_step.bat	19/10/2021 12:22

After finishing this we'll set up the config file required for the Microsoft Store.

Sandbox Get/Set

When you are using Xbox Live features you might need to change your PC sandboxes (see **official documentation**) so you can test those same features. For this purpose, run the GDK Command Line (under `Start → All Apps → Microsoft GDK → Desktop VS 2018 Gaming Command Prompt`) and use one of the following commands:

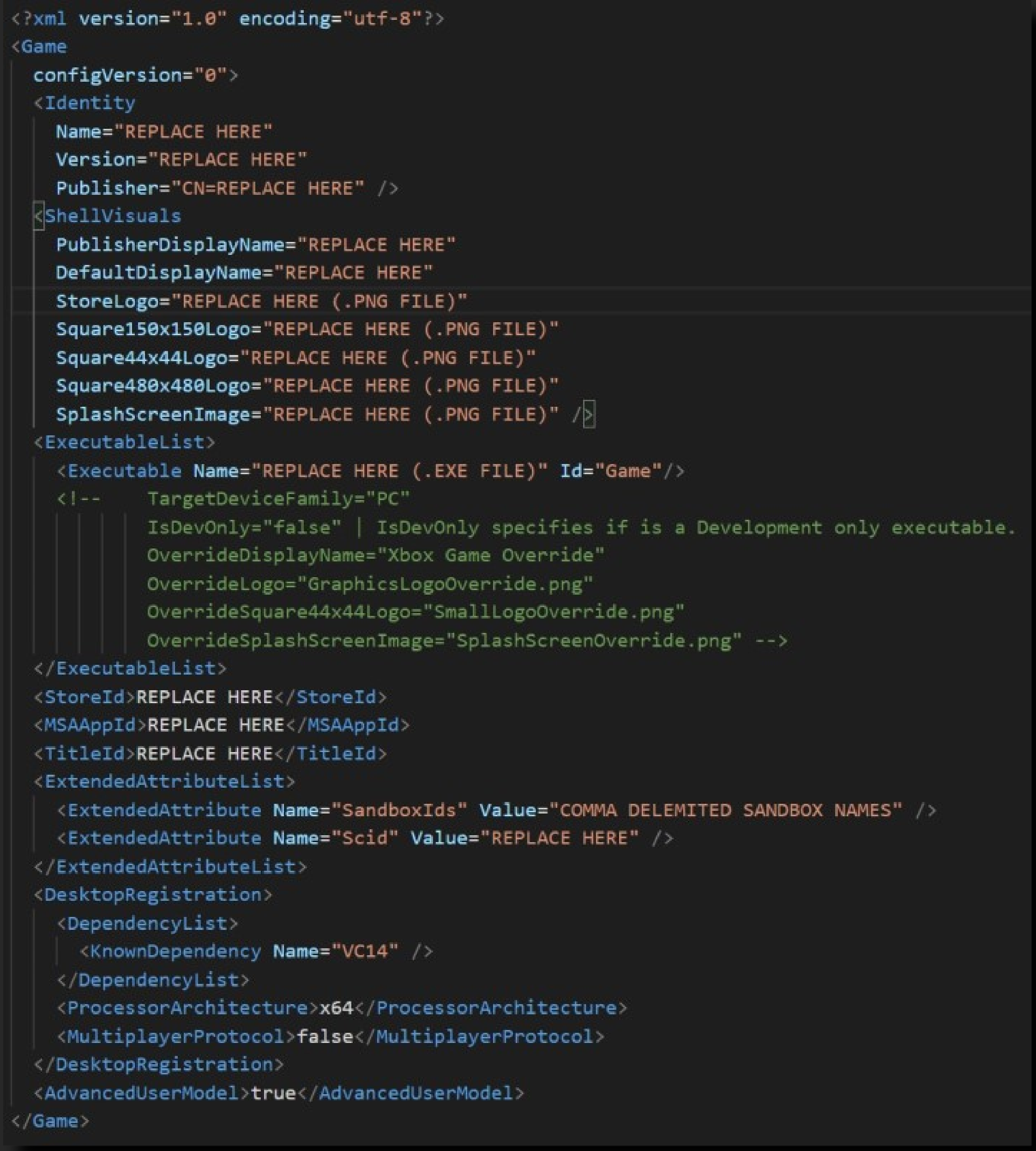
- `XblPCSandbox /get` (returns the current sandbox, default is RETAIL)
- `XblPCSandbox <sandbox>` (where `<sandbox>` refers to you sandbox name)
- `XblPCSandbox RETAIL` (sets the sandbox back to RETAIL)

NOTE Sandbox names are case sensitive

Config File

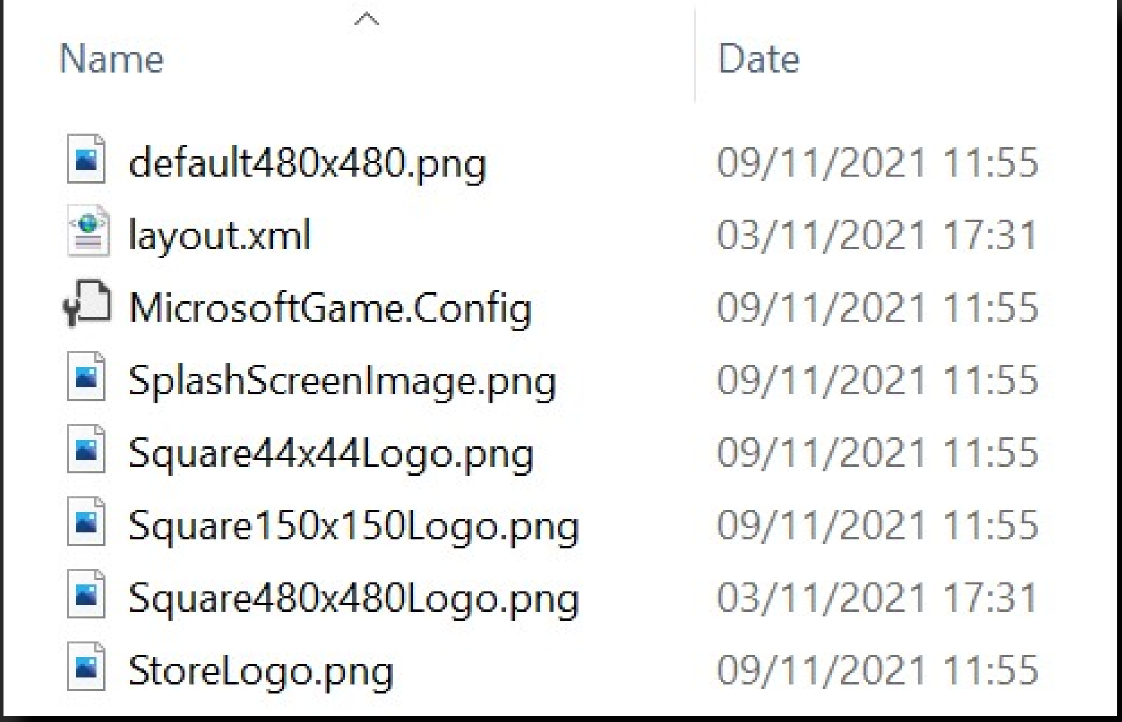
To build and run using GDK Extension it' necessary to set up and include a `MicrosoftGame.Config` file:

1. This config file can be created using Microsoft's MicrosoftGame Editor tool (for more information refer to its **documentation page**) or can be copied from the **demo project** and edited manually.


```xml
<?xml version="1.0" encoding="utf-8"?>
<Game
  configVersion="0">
  <Identity
    Name="REPLACE HERE"
    Version="REPLACE HERE"
    Publisher="CN=REPLACE HERE" />
  <ShellVisuals
    PublisherDisplayName="REPLACE HERE"
    DefaultDisplayName="REPLACE HERE"
    StoreLogo="REPLACE HERE (.PNG FILE)"
    Square150x150Logo="REPLACE HERE (.PNG FILE)"
    Square44x44Logo="REPLACE HERE (.PNG FILE)"
    Square480x480Logo="REPLACE HERE (.PNG FILE)"
    SplashScreenImage="REPLACE HERE (.PNG FILE)" />
  <ExecutableList>
    <Executable Name="REPLACE HERE (.EXE FILE)" Id="Game"/>
    <!--    TargetDeviceFamily="PC"
            IsDevOnly="false" | IsDevOnly specifies if is a Development only executable.
            OverrideDisplayName="Xbox Game Override"
            OverrideLogo="GraphicsLogoOverride.png"
            OverrideSquare44x44Logo="SmallLogoOverride.png"
            OverrideSplashScreenImage="SplashScreenOverride.png" -->
  </ExecutableList>
  <StoreId>REPLACE HERE</StoreId>
  <MSAAppId>REPLACE HERE</MSAAppId>
  <TitleId>REPLACE HERE</TitleId>
  <ExtendedAttributeList>
    <ExtendedAttribute Name="SandboxIds" Value="COMMA DELEMITED SANDBOX NAMES" />
    <ExtendedAttribute Name="Scid" Value="REPLACE HERE" />
  </ExtendedAttributeList>
  <DesktopRegistration>
    <DependencyList>
      <KnownDependency Name="VC14" />
    </DependencyList>
    <ProcessorArchitecture>x64</ProcessorArchitecture>
    <MultiplayerProtocol>false</MultiplayerProtocol>
  </DesktopRegistration>
  <AdvancedUserModel>true</AdvancedUserModel>
</Game>
```

NOTE The `ExecutableList/Executable/Name` property should be set to the value in `GameOptions → Windows → Executable Name`.

2. Add the `MicrosoftGame.Config` file to the **Included Files** of your project.
3. Depending on the image file names specified in the `ShellVisuals` tag you will need to add those to the **Included Files** as well.



Name	Date
default480x480.png	09/11/2021 11:55
layout.xml	03/11/2021 17:31
MicrosoftGame.Config	09/11/2021 11:55
SplashScreenImage.png	09/11/2021 11:55
Square44x44Logo.png	09/11/2021 11:55
Square150x150Logo.png	09/11/2021 11:55
Square480x480Logo.png	03/11/2021 17:31
StoreLogo.png	09/11/2021 11:55

Upon finishing this setup and following the **Project Setup** section above, you should be ready to run and test your project.

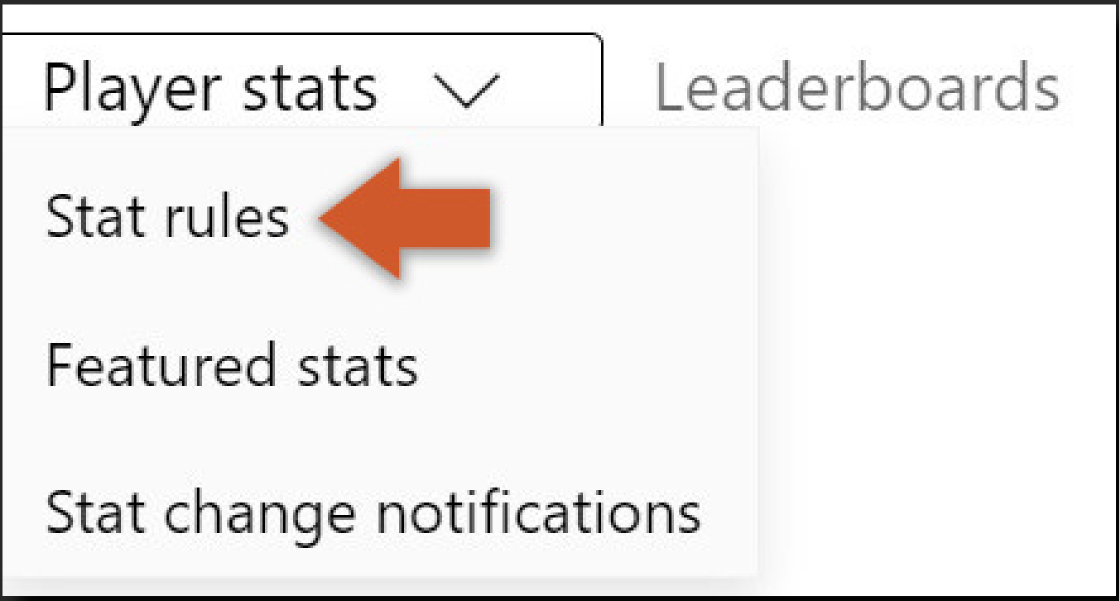
Manifest File

This step is not required unless you are using the **Event Based Functions** from the Xbox Live module (this requires enabling of **Event-Based Stats** on the Partner Center).

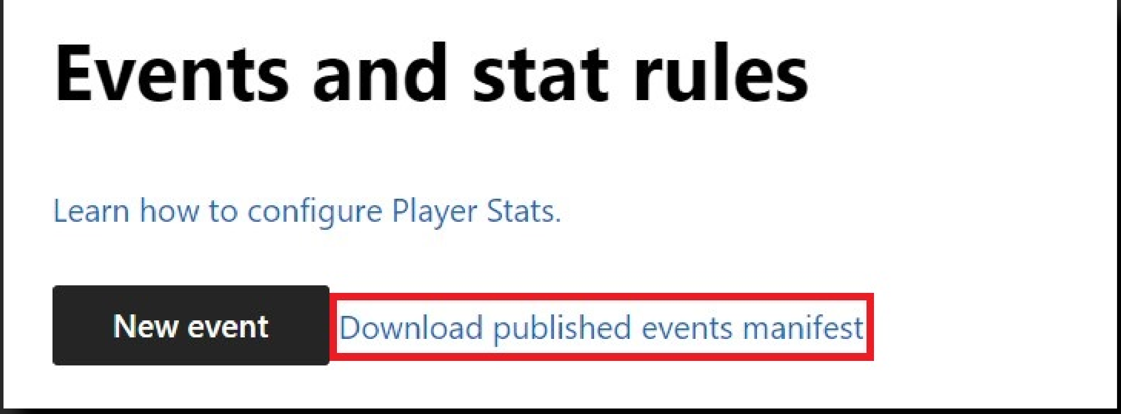
If you are using the event based system from Xbox Live module you need provide GameMaker Studio 2 with some extra information about the **statistics**, **events** and **leaderboards** created on you Partner Center's dashboard. For this you need to do the following:

1. Log into your **Partner Center** account, open your game's configuration page and go into the **Xbox Services → Gameplay Settings** where you can define your stats.

2. From here you will need to select **Player stats → Stat rules** the drop-down menu at the top.



3. On the page that opens, click on the "**Download published events manifest**" button.


Events and stat rules

Learn how to configure Player Stats.

4. Add the downloaded file to your project's **Included Files (/datafiles)** folder.

After setting up this manifest file, you will be able to use event-based stats and leaderboards in your project.

> **IMPORTANT** If you make any changes to your events on the Partner Center dashboard, you'll need to republish those changes and download a new **manifest** file.

GDK Extension API

This is a collection of GameMaker Studio functions for interacting with the GDK Extension API. This API is comprised of modules that will allow you to access multiple Xbox Live features such as profile information, stats, leaderboards, achievements, storage and Microsoft Store add-ons (also known as In-App Purchases).

Management

Using the GDK Extension **requires** the developer to manually manage it. The following functions are provided for managing the extension:

- gdk_init
- gdk_update
- gdk_quit

Modules

Please look at the following sections for information on the different modules present in this library:

- Base Module (User)
- Storage Module (Save & Load)
- Xbox Live Module (Stats & Achievements)
- IAP Module (In-App Purchases)

gdk_init

This function must be called before any other GDK extension function. It is recommended to use a persistent controller/manager object that is created or placed in the first room and to then call this function in its Create Event.

Syntax:

```
gdk_init(scid);
```

Argument	Type	Description
scid	string	The service configuration ID (found under **Game setup** on the Partner Center).

Returns:

```
N/A
```

Example:

```
gdk_init("00000000-0000-0000-0000-000060ddb039");
```

The code above initialises the GDK Extension with the provided **SCID** string. This value can be acquired from your `MicrosoftGame.Config` file (that needs to be placed in the included files) or from the Partner Center.

gdk_update

This should be called each frame while GDK extension is active, recommend using a Controller persistent object that is created or placed in the first room and this call is in the **Step Event**.

Syntax:

```
gdk_update();
```

Returns:

```
N/A
```

Example:

```
gdk_update();
```

In the code above we are ticking the update logic of the GDK extension, faling to call this function will make the extension stop working.

gdk_quit

This should be called on close down and no GDK extension functions should be called after it, recommend using a Controller (persistent) object that is created or placed in the first room and this call is in its **Destroy Event**.

Syntax:

```
gdk_quit();
```

Returns:

```
N/A
```

Example:

```
gdk_quit();
```

In the code above will terminate the GDK extension and no GDK extension functions should be called after it.

Base Module

The Base Module in the GDK Extension provides a set of functions to handle the user/ account selection for the player.

Base Functions

The following functions are given for working with user accounts:

- xboxone_show_account_picker
- xboxone_get_activating_user
- xboxone_get_user_count
- xboxone_get_user

xboxone_show_account_picker

With this function you can retrieve the user ID pointer for the user that launched the game.

> **NOTE** The returned user ID may not always be accurate as the active account can be changed at any time and this value will always return the user account that was active when the game launched.

Syntax:

```
xboxone_get_activating_user();
```

Returns:

```
Pointer (The user ID pointer)
```

Example:

```
global.main_user = xboxone_get_activating_user();
```

In the code above we get the ID of the user responsible for launching the game and store it inside a global variable (`global.main_user`).

xboxone_get_user

With this function you can retrieve the user ID pointer for the indexed user. If the user does not exist, the function will return the constant `pointer_null` instead. You can find the number of users currently logged in with the function **xboxone_get_user_count**.

Syntax:

```
xboxone_get_user(index);
```

Argument	Type	Description
index	integer	The index to get the user ID from.

Returns:

```
Pointer (The user ID pointer)
```

Example:

```
for (var i = 0; i < xboxone_get_user_count(); i++)
{
    user_id[i] = xboxone_get_user(i);
}
```

In the code above we loop through all the users currently singed in (using the `xboxone_get_user_count` function) and store their user IDs into an array (`user_id`).

xboxone_get_user_count

With this function you can find the total number of users currently signed in to the system. The returned value will be an integer.

Syntax:

```
xboxone_get_user_count();
```

Returns:

```
Real (The total number of users currently signed in to the system)
```

Example:

```
for (var i = 0; i < xboxone_get_user_count(); i++)
{
    user_id[i] = xboxone_get_user(i);
}
```

In the code above we loop through all the users currently singed in to the system and store their user IDs into an array (using the `xboxone_get_user` function).

xboxone_show_account_picker

This function launches the system's account picker for the specified pad ID, where selecting an account will associate it with the specified pad. The `guests` argument is either `true` or `false` – if `false` is specified no guest accounts can be selected, while `true` allows guest accounts.

IMPORTANT The argument `pad_id` is ignored by the GDK Extension and is only used when the function is called on console.

This is an asynchronous function that will trigger the **Async Dialog** event when the task is finished.

Syntax:

```
xboxone_show_account_picker(pad_id, guests);
```

Argument	Type	Description
pad_id	real	**UNUSED** Pad ID of the user to select the account for
guests	boolean	Whether we should allow guest accounts

Returns:

```
Real (The asynchronous Request ID)
```

Triggers:

```
Asynchronous Dialog Event
```

Key	Type	Description
id	real	The asynchronous request ID.
type	string	The string `"xboxone_accountpicker"`

succeeded	boolean	Returns `true` if an account was select, `false` if cancelled
user	pointer	The store ID used when acquiring the license.

Example:

```
requestId = xboxone_show_account_picker(0, false);
```

In the code above we first request for an account selection dialog that allows no guest accounts. The function call will then return a request ID (`requestId`) that can be used inside an Async Dialog event.

```
if (async_load[? "id"] == requestId)
{
    if (async_load[? "succeeded"] == false)
    {
        show_debug_message("User cancelled account selection!");
    }
}
```

The code above matches the response against the correct event **id**, providing a failure message if `"succeeded"` is `false`.

Storage Module

The Storage Module provides the developer with functions to save and load data associated with a given user.

Storage Functions

The following functions are provided for working with user data storage:

- xboxone_get_savedata_user
- xboxone_set_savedata_user
- gdk_save_group_begin
- gdk_save_buffer
- gdk_save_group_end
- gdk_load_buffer

gdk_load_buffer

This function is used to load data from a file that you previously saved using the **gdk_save_buffer** function. The data from the file is loaded into a buffer that you need to have created previously.

The `offset` defines the starting position within the buffer for loading (in bytes), and the `size` is the size of the buffer area to be loaded from that offset onwards (also in bytes). You can supply a value of -1 for the size argument and the entire buffer will be loaded.

Note that the function will load from a "default" folder, which does not need to be included as part of the file path you provide. This folder will be created if it doesn't exist when you save a file using **gdk_save_buffer**.

IMPORTANT Before using this function it's required to set the workspace for storage operations using the function **xboxone_set_savedata_user**.

This is an asynchronous function that will trigger the **Async Save/Load** event when the task is finished.

Syntax:

```
gdk_load_buffer(buffer_idx, filename, offset, size);
```

Argument	Type	Description
buffer_idx	integer	The index of the buffer to load data into.
filename	string	The name of the file to load.
offset	integer	The offset within the buffer to load from (in bytes).
size	integer	The size of the buffer area to load (in bytes).

Returns:

```
Real (-1 if there was an error, otherwise the task request ID)
```

Triggers:

```
Asynchronous Save/Load Event
```

Key	Type	Description
id	integer	The unique identifier of the asynchronous request.
error	real	0 is successful, some other value if there has been as error (error code).
status	real	1 if successful, 0 if failed.
file_size	real	The total size of the file being loaded.
load_size	real	The amount of bytes loaded into the buffer.

Example:

```
requestId = gdk_load_buffer(buff, "Player_Save.sav", 0, 16384);
```

In the code above we load a file into a buffer (`buff`). The function call will then return a request ID (`requestId`) that can be used inside an Async Save/Load event.

```
if (async_load[? "id"] == requestId)
{
    if (async_load[? "status"] == false)
    {
        show_debug_message("Load failed!");
    }
    else
    {
        show_debug_message("Load succeeded!");
    }
}
```

The code above matches the response against the correct request **id**, providing a success message if **status** is true.

gdk_save_buffer

With this function you can save a buffer (or a part of it) to a file, ready to be read back into memory using the gdk_load_buffer function. The `offset` defines the start position within the buffer to start saving from (in bytes), and the `size` is the size of the buffer area to be saved from that offset onwards (also in bytes).

All files saved using this function will be placed in a "default" folder. This folder does not need to be included in the file path as it is added automatically by GameMaker. For example the filename path `"Data\Player_Save.sav"` would actually be saved to `"default\Data\Player_Save.sav"`. However, if you then load the file using the function gdk_load_buffer, you do not need to supply the "default" part of the path either.

IMPORTANT Before using this function it's required to set the workspace for storage operations using the function xboxone_set_savedata_user.

This is an asynchronous function that will trigger the **Async Save/Load** event when the task is finished.

Syntax:

```
gdk_save_buffer(buffer_idx, filename, offset, size);
```

Argument	Type	Description
buffer_idx	integer	The index of the buffer to save.
filename	string	The place where to save the buffer to (path + filename + extension).
offset	integer	The start position within the buffer for saving (in bytes).
size	integer	The size of the buffer area to be saved (in bytes).

Returns:

```
Real (-1 if there was an error, otherwise the task request ID)
```

Triggers:

```
Asynchronous Save/Load Event
```

Key	Type	Description
id	integer	The unique identifier of the asynchronous request.
error	real	0 is successful, some other value if there has been as error (error code).
status	real	1 if successful, 0 if failed.

Example:

```
requestId = gdk_save_buffer(buff, "Player_Save.sav", 0, 16384);
```

In the code above we save a buffer (`buff`) into a file. The function call will then return a request ID (`requestId`) that can be used inside an Async Save/Load event.

```
if (async_load[? "id"] == requestId)
{
    if (async_load[? "status"] == false)
    {
        show_debug_message("Save failed!");
    }
    else
    {
        show_debug_message("Save succeeded!");
    }
}
```

The code above matches the response against the correct request **id**, providing a success message if **status** is true.

gdk_save_group_begin

This function is called when you want to begin the saving of multiple buffers to multiple files. The `container_name` is a string and will be used as the directory name for where the files will be saved, and should be used as part of the file path when loading the files back into the IDE later (using the **gdk_load_buffer** function). This function is only for use with the **gdk_save_buffer** function and you must also end the group save by calling **gdk_save_group_end** function otherwise the files will not be saved out.

This is an asynchronous function that will trigger the **Async Save/Load** event when the task is finished.

Syntax:

```
gdk_save_group_begin(container_name);
```

Argument	Type	Description
container_name	string	The name of the container.

Returns:

```
N/A
```

Example:

```
gdk_save_group_begin("SaveGame");
save1 = gdk_save_buffer(buff1, "Player_Save1.sav", 0, 16384);
save2 = gdk_save_buffer(buff2, "Player_Save2.sav", 0, 16384);
save3 = gdk_save_buffer(buff3, "Player_Save3.sav", 0, 16384);
save4 = gdk_save_buffer(buff4, "Player_Save4.sav", 0, 16384);
gdk_save_group_end();
```

In the code above we save multiple buffers into different files using a buffer group for them. All the files will be saved inside the same `"SaveGame"` folder.

gdk_save_group_end

This function finishes the definition of a buffer save group. You must have previously called the function gdk_save_group_begin to initiate the group, then call the function gdk_save_buffer for each file that you wish to save out. Finally you call this function, which will start the saving of the files.

This is an asynchronous function that will trigger the **Async Save/Load** event when the task is finished.

Syntax:

```
gdk_save_group_end();
```

Returns:

```
Real (-1 if there was an error, otherwise the task request ID)
```

Triggers:

```
Asynchronous Save/Load Event
```

Key	Type	Description
id	integer	The unique identifier of the asynchronous request.
error	real	0 is successful, some other value if there has been as error (error code).
status	real	1 if successful, 0 if failed.

Example:

```
gdk_save_group_begin("SaveGame");
gdk_save_buffer(buff1, "Player_Save1.sav", 0, 16384);
gdk_save_buffer(buff2, "Player_Save2.sav", 0, 16384);
gdk_save_buffer(buff3, "Player_Save3.sav", 0, 16384);
gdk_save_buffer(buff4, "Player_Save4.sav", 0, 16384);
requestId = gdk_save_group_end();
```

In the code above we save multiple buffers into different files using a buffer group for them. All the files will be saved inside the same `"SaveGame"` folder. Afterwards we end the group with a `gdk_save_group_end()` call, which will then return a requestID (`requestId`) that can be used inside an Async Save/Load event.

```
if (async_load[? "id"] == requestId)
{
    if (async_load[? "status"] == false)
    {
        show_debug_message("Group save failed!");
    }
    else
    {
        show_debug_message("Group save succeeded!");
    }
}
```

The code above matches the response against the correct request **id**, providing a success message if **status** is true.

xboxone_get_savedata_user

This function returns the user ID pointer (or the constant `pointer_null`) currently associated with file saving. See xboxone_set_savedata_user for further details.

Syntax:

```
xboxone_get_savedata_user();
```

Returns:

```
Pointer (The user ID currently being used for save data)
```

Example:

```
if (xboxone_get_savedata_user() != user_id[0])
{
    xboxone_set_savedata_user(user_id[0]);
}
```

In the code above we check to see if the current savedata user is set to the user in the index 0 of the `user_id` array, and if not, we set it to that user id.

xboxone_set_savedata_user

This function sets all future file operations operating in the save game area (i.e. all file reads and writes made using the **gdk_load_buffer** and **gdk_save_buffer** functions) to be associated with the specified user. This can be called as often as necessary to redirect save data to the appropriate user, or you can use the constant `pointer_null` to lock save/load features.

Syntax:

```
xboxone_set_savedata_user(user_id);
```

Argument	Type	Description
user_id	pointer	The user ID pointer.

Returns:

```
N/A
```

Example:

```
if (xboxone_get_savedata_user() != user_id[0])
{
    xboxone_set_savedata_user(user_id[0]);
}
```

In the code above we check to see if the current savedata user is set to the user in the index 0 of the `user_id` array, and if not, we set it to that user id.

Xbox Live Module

The GDK extension allows your Microsoft Store game to use all features from the Xbox Live services including stats, leaderboards, achievements and rich presence. This module contains all the functions that are available to handle Xbox Live services.

Xbox Live services can either be E**vent-Based** or T**itle-Managed and this choice is made on the Partner Center** (check the help article on the Partner Center for more information). The functions below are grouped according these two systems.

Event Based Functions

IMPORTANT Using the Event-Based system requires some additional configuration detailed under Manifest File guide.

The following functions are provided for event based stats/leaderboards/achievements (Microsoft recommends using these for **stats** and **leaderboards** but not **achievements**):

- xboxone_stats_setup
- xboxone_check_privilege
- xboxone_fire_event
- xboxone_read_player_leaderboard

Title Managed Functions

The following functions are provided for title managed stats/leaderboards/achievements (Microsoft recommends using these for **achievements**):

- xboxone_stats_add_user
- xboxone_stats_remove_user
- xboxone_stats_flush_user
- xboxone_stats_get_stat
- xboxone_stats_get_stat_names

- xboxone_stats_set_stat_int
- xboxone_stats_set_stat_real
- xboxone_stats_set_stat_string
- xboxone_stats_delete_stat
- xboxone_stats_get_leaderboard
- xboxone_stats_get_social_leaderboard
- xboxone_achievements_set_progress

General Functions

The following functions can be used regardless of whether Event-Based or Title-Managed services are used:

- xboxone_set_rich_presence

xboxone_achievements_set_progress

This function can be used to update the progress of an achievement. You supply the `user_id` as returned by the function **xboxone_get_user**, a string containing the `achievement`'s numeric ID (as assigned in the Partner Center when it was created), and finally the `progress` value to set (from 0 to 100).

IMPORTANT Note that the achievement system will refuse updates that are lower than the current progress value.

This is an asynchronous function that will trigger the **Async System** event when the task is finished.

Syntax:

```
xboxone_achievements_set_progress(user_id, achievement, progress);
```

Argument	Type	Description
user_id	pointer	The user ID pointer.
achievement	string	The achievement identification string (obtained from the Partner Center).
progress	real	The new progress value of the achievement (is a value between 0 and 100).

Returns:

```
Real (negative if error, otherwise the System Request ID)
```

Error	Description
-1	Invalid `user_id` was specified.
-2	Xbox Live context could not be retrieved.

Triggers:

```
Asynchronous System Event
```

Key	Type	Description
event_type	string	The string `"achievement result"`
requestID	real	The id of the request that fired this callback.
achievement	string	The achievement ID string passed to the function call.
progress	real	The updated progress value.
error	real	This will be: `0` if the progress update was successful; `xboxone_achievement_already_unlocked` if the achievement was unlocked in a previous request; or a negative number with the error code if request fails.

Example:

```
requestId = xboxone_achievements_set_progress(user_id, "KilledFirstBoss", 100);
```

In the code above we setting the `"KilledFirstBoss"` achievement as 100% complete, tshe function call will then return a request ID (`requestId`) that can be used inside an Async System event.

```
if (async_load[? "event_type"] == "achievement_result")
{
    if (async_load[? "requestID"] == requestId)
    {
        if (async_load[? "error"] >= 0)
        {
            show_debug_message("Request succeeded");
        }
    }
}
```

The code above matches the response against the correct **event_type** and **requestID**, and prints a debug message if the request was successful.

xboxone_check_privilege

With this function you can check whether the given user has a privilege. If you set the `attempt_resolution` argument to `true` and the privilege isn't enabled, it will also open a system dialogue (suspending the game) to prompt the user to upgrade their account or to get the privilege in a different way as required. If the user then acquires the required privilege, the function will return `true`.

This is an asynchronous function that will trigger the **Async System** event when the task is finished.

Syntax:

```
xboxone_check_privilege(user_id, privilege_id, attempt_resolution);
```

Argument	Type	Description
user_id	pointer	The user ID pointer.
privilege_id	constant	The privilege to check for (is a `xboxone_privilege_*` constant)
attempt_resolution	boolean	Requests for this privilege.

Returns:

```
N/A
```

Triggers:

```
Asynchronous System Event
```

Key	Type	Description
event_type	string	The string `"check_privilege_result"`.
result	integer	One or more (bit-wise combined) of the `xboxone_privilege_*` constants.
privilege	constant	The privilege you have requested (is a `xboxone_privilege_*` constant)

Example:

```
var user_one = xboxone_get_activating_user();
xboxone_check_privilege(user_one, xboxone_privilege_multiplayer_sessions, true);
```

In the code above we are getting the ID of the user that launched the game (using the function **xboxone_get_activating_user**) and checking if they have privilege for multiplayer sessions and requesting for attempting resolution.

xboxone_fire_event

This function can be used to fire a stat event. The `event_name` argument is the name of the event to be fired as defined in the Partner Center console for your game, and the following additional parameters will also depend on what you have a set up for the stat.

The function will return 0 if the event was sent (and should be received/processed by the server) or -1 if there was an error (e.g. your events manifest file is outdated).

> **NOTE** The first two parameters for an event usually default to the `user_id` (obtained from the **xboxone_get_user** function) and the game's `scid` (obtained from the Microsoft Partner Center).

Syntax:

```
xboxone_fire_event(event_name, ...);
```

Argument	Type	Description
event_name	string	The name of the event to be triggered.
...	*	The parameters to be passed to the event as individual arguments.

Returns:

```
Real (-1 on error, 0 if the function was successful)
```

Example:

```
var uid = xboxone_get_activating_user();
var result = xboxone_fire_event("PlayerSessionStart", uid, global.scid, 0, 42, 42);
```

In the code above we are firing the `"PlayerSessionStart"` event and passing some parameters, including the player's ID and the game's SCID as the first two parameters for the specified event.

xboxone_read_player_leaderboard

This function allows you to read a leaderboard starting at the specified user, regardless of the user's rank or score, and ordered by percentile rank. You supply the `user_id` as returned by the function **xboxone_get_user** and a `friendfilter` that is one of the `achievement_filter_*` constants.

IMPORTANT This function requires **xboxone_stats_setup** before it can be used.

This is an asynchronous function that will trigger the **Async Social** event when the task is finished.

Syntax:

```
xboxone_read_player_leaderboard(ident, user_id, numitems, friendfilter);
```

Argument	Type	Description
ident	string	The leaderboard id (if filter is `all_players`) or stat to read.
user_id	pointer	The user ID pointer.
numitems	real	The number of items to retrieve.
friendfilter	constant	One of the `achievement_filter_*` constants.

Returns:

```
Real (-1 on error, any other value otherwise)
```

Triggers:

```
Asynchronous Social Event
```

Key	Type	Description
id	constant	The constant `achievement_leaderboard_info`

leaderboardid	string	The unique ID of the leaderboard as defined on the provider dashboard.
numentries	real	The number of entries in the leaderboard that you have received.
PlayerN	string	The name of the player, where **N** is position within the received entries list.
PlayeridN	pointer	The unique user id of the player **N**
RankN	real	The rank of the player **N** within the leaderboard.
ScoreN	real	The score of the player **N**

Example:

```
var uid = xboxone_get_activating_user();
xboxone_read_player_leaderboard("MyLeaderboard", uid, 10,
achievement_filter_all_players);
```

In the code above we are querying the leaderboard with the ID `"MyLeaderboard"` for the first 10 entries including all players.

We can catch the triggered callback using the the Async Social event.

```
if (async_load[? "id"] == achievement_leaderboard_info)
{
    global.numentries = async_load[? "numentries"];
    for (var i = 0; i < numentries; i++)
    {
        global.playername[i] = async_load[? "Player" + string(i)];
        global.playerid[i] = async_load[? "Playerid" + string(i)];
        global.playerrank[i] = async_load[? "Rank" + string(i)];
        global.playerscore[i] = async_load[? "Score" + string(i)];
    }
}
```

The code above matches the response against the correct **id** and stores the information of the top player names, ids, ranks and scores in global arrays.

xboxone_set_rich_presence

This function is used to set the rich presence string for the given user. A Rich Presence string shows the user's in-game activity after the name of the game that the user is playing, separated by a hyphen. This string is displayed under a player's Gamertag in the "Friends & Clubs" list as well as in the player's Xbox Live profile.

When using this function you need to supply the `user_id` for the user, and then you can flag the user as currently active in the game or not (using `true` / `false`). The next argument is the `rich_presence_string` ID to show, and then finally you can (optionally) supply a `scid` string. Note that this is an optional argument -- if you have called **xboxone_stats_setup** you don't need to pass the `scid` here.

> **TIP** For more information on rich presence and how to set up the strings to use in the Partner Center, please see the Microsoft's **Rich Presence** documentation.

> **IMPORTANT** On the Windows platform it is not possible to check your rich presence string during development (when it's a sandboxed project).

Syntax:

```
xboxone_set_rich_presence(user_id, is_user_active, rich_presence_id, scid);
```

Argument	Type	Description
user_id	pointer	The user ID pointer
is_user_active	boolean	Flag the user as active or not
rich_presence_id	string	The rich present ID (defined in the Partner Center)
scid	string	OPTIONAL The Service Configuration ID string

Returns:

```
N/A
```

Example:

```
var user_one = xboxone_get_activating_user();
xboxone_set_rich_presence(userId, true, "Playing_Challenge", global.scid);
```

In the code above we are getting the active user (using the function **xboxone_get_activating_user**) and setting its current presence string. We are providing a scid (`global.scid`) otherwise we were required to first call the function **xboxone_stats_setup**.

xboxone_stats_setup

This function needs to be called before you can use any of the other Xbox stat functions, and simply initializes the required libraries on the system. The `user_id` argument is the user ID as returned by the function **xboxone_get_user**, while the `scid` and `title_id` are the unique ID's for your game on the Microsoft Partner Center.

Syntax:

```
xboxone_stats_setup(user_id, scid, title_id);
```

Argument	Type	Description
user_id	pointer	The user ID pointer.
scid	string	The Service Configuration ID (SCID)
title_id	integer (hex)	The title ID (as shown in `"MicrosoftGame.Config"`)

Returns:

```
N/A
```

Example:

```
var user_id = xboxone_get_user(0);
xboxone_stats_setup(user_id, "00000000-0000-0000-0000-000000000000", 0xFFFFFFFF);
```

In the code above first we retrieve the user ID of the user with index 0 (using the **xboxone_get_user** function) and use it to setup the event based stat system.

xbox_stats_add_user

This function can be used to add the given user to the statistics manager. This must be done before using any of the other stats functions to automatically sync the game with the Xbox Live server and retrieve the latest values. You supply the `user_id` as returned by the function **xboxone_get_user**, and the function will return -1 if there was an error or the `user_id` is invalid, or an async request ID if the function was successfully called.

This is an asynchronous function that will trigger the **Async Social** event when the task is finished.

Syntax:

```
xboxone_stats_add_user(user_id);
```

Argument	Type	Description
user_id	pointer	The user ID pointer.

Returns:

```
Real (-1 on error, otherwise an async request ID)
```

Triggers:

```
Asynchronous Social Event
```

Key	Type	Description
id	constant	The constant `achievement_stat_event`
event	string	The string `"LocalUserAdded"`.
userid	pointer	The user ID associated with the request.
error	real	0 if successful, some other value on error.
errorMessage	string	OPTIONAL A string with an error message

Example:

```
for(var i = 0; i < xboxone_get_user_count(); i++)
{
    user_id[i] = xboxone_get_user(i);
    xboxone_stats_add_user(user_id[i]);
}
```

In the code above we are looping though all the available users (using the function **xboxone_get_user_count**) getting their user id (using the **xboxone_get_user** function) and adding them to the statistics manager.

We can catch the triggered callbacks using the the **Async Social** event.

```
if (async_load[? "event"] == "LocalUserAdded")
{
    if (async_load[? "error"] != 0)
    {
        show_debug_message("Error while adding user to statistics manager");
    }
}
```

The code above matches the response against the correct **event**, providing a failure message if **error** is not 0.

xboxone_stats_delete_stat

This function can be used to delete a stat from the stat manager for the given user ID. You supply the user ID as returned by the function **xboxone_get_user**, then the stat string to delete. This clears the stat value and removed it from the stat manager, meaning it will no longer be returned by the functions **xboxone_stats_get_stat_names** and **xboxone_stats_get_stat**.

Syntax:

```
xboxone_stats_delete_stat(user_id, stat_name)
```

Argument	Type	Description
user_id	pointer	The user ID pointer.
stat_name	string	The stat to be deleted.

Returns:

```
Real (-1 on error, any other value otherwise)
```

Example:

```
for (var i = 0; i < xboxone_get_user_count(); i++)
{
    user_id[i] = xboxone_get_user(i);
    xboxone_stats_delete_stat(user_id[i], "highScore");
}
```

In the code above we are looping though all the available users (using the function **xboxone_get_user_count**) getting their user id (using the **xboxone_get_user** function) and deleting the stat `"highScore"` from each one of them.

xbox_stats_flush_user

This function can be used to flush the stats data for the given user from the statistics manager to the live server, ensuring that the server is up to date with the current values. To use the function, you supply the `user_id` as returned by the function **xboxone_get_user**, and then give a priority value (0 for low priority and 1 for high priority). The function will will return -1 if there was an error or the user ID was invalid, or an async request ID if the function was successfully called.

> **IMPORTANT** According to Xbox documentation, developers should be careful not to call this too often as the Xbox system will rate-limit the requests, and the GDK Extension will also automatically flush stats approximately every 5 minutes.

This is an asynchronous function that will trigger the **Async Social** event when the task is finished.

Syntax:

```
xboxone_stats_flush_user(user_id, high_priority);
```

Argument	Type	Description
user_id	pointer	The user ID pointer.
high_priority	boolean	Whether or not flush is high priority.

Returns:

```
Real (-1 on error, otherwise an async request ID)
```

Triggers:

```
Asynchronous Social Event
```

Key	Type	Description
id	constant	The constant `achievement_stat_event`
event	string	The string `"StatisticUpdateComplete"`.

userid	pointer	The user ID associated with the request.
error	real	0 if successful, some other value on error.
errorMessage	string	A string with an error message **[OPTIONAL]**

Example:

```
for(var i = 0; i < array_length(user_ids); i++)
{
    xboxone_stats_flush_user(user_ids[i]);
}
```

In the code above we are looping though an array of user ids (`user_ids`) and flushing their local data to the live server.

We can catch the triggered callbacks using the the **Async Social** event.

```
if (async_load[? "event"] == "StatisticsUpdateComplete")
{
    if (async_load[? "error"] != 0)
    {
        show_debug_message("Error while updating user statistics");
    }
}
```

The code above matches the response against the correct **event**, providing a failure message if **error** is not 0.

xboxone_stats_get_leaderboard

This function can be used to retrieve a global leaderboard with ranks for a given statistic. You supply the user ID (as returned by the function xboxone_get_user), the stat string (as defined when you registered it as a "Featured Stat"), and then you specify a number of details about what leaderboard information you want to retrieve. Note that you can only retrieve a global leaderboard for `int` or `real` stats, but not for `string` stats.

> IMPORTANT Stats used in global leaderboards must be registered as "Featured Stats" in the Partner Center otherwise an error will be returned. If you want local (social) leaderboards, then please see the function xboxone_stats_get_social_leaderboard.

> IMPORTANT The user ID passed into this function should first be added to the statistics manager using the xboxone_stats_add_user function.

This is an asynchronous function that will trigger the **Async Social** event when the task is finished.

Syntax:

```
xboxone_stats_get_leaderboard(user_id, stat, num_entries, start_rank, start_at_user,
ascending);
```

Argument	Type	Description
user_id	pointer	The user ID pointer.
stat	string	The stat to create the global leaderboard from.
num_entries	real	The number of entries from the leaderboard to retrieve.
start_rank	real	The rank in the leaderboard to start from (set to `0` if `start_at_user` is set to true.
start_at_user	boolean	Set to `true` to start at the user ID.
ascending	boolean	Set to `true` or ascending or `false` for descending order.

Returns:

```
N/A
```

Triggers:

```
Asynchronous Social Event
```

Key	Type	Description
id	constant	The constant `achievement_leaderboard_info`
event	string	The string `"GetLeaderboardComplete"`.
userid	pointer	The user ID associated with the request.
error	real	0 if successful, some other value on error.
errorMessage	string	A string with an error message **[OPTIONAL]**
displayName	string	The unique ID for the leaderboard as defined on the provider dashboard.
numentries	real	The number of entries in the leaderboard that you have received.
PlayerN	string	The name of the player, where **N** is position within the received entries list.
PlayeridN	pointer	The unique user id of the player **N**
RankN	real	The rank of the player **N** within the leaderboard.
ScoreN	real	The score of the player **N**

Example:

```
xboxone_stats_get_leaderboard(user_id, "GlobalTime", 20, 1, false, true);
```

In the code above we are querying a leaderboard for the stat `"GlobalTime"` with the first 20 entries starting on rank 1 in ascending order.

We can catch the triggered callback using the the **Async Social** event.

```
if (async_load[? "id"] == achievement_leaderboard_info)
{
    if (async_load[? "event"] == "GetLeaderboardComplete")
    {
        global.numentries = async_load[? "numentries"];
        for (var i = 0; i < numentries; i++)
        {
            global.playername[i] = async_load[? "Player" + string(i)];
            global.playerid[i] = async_load[? "Playerid" + string(i)];
            global.playerrank[i] = async_load[? "Rank" + string(i)];
            global.playerscore[i] = async_load[? "Score" + string(i)];
        }
    }
}
```

The code above matches the response against the correct **id** and **event**, and stores the information of the top player names, ids, ranks and scores in global arrays.

xboxone_stats_get_social_leaderboard

This function can be used to retrieve a social leaderboard with ranks for a given statistic. You supply the user ID (as returned by the function **xboxone_get_user**), the stat string (as defined when you created it using the `xboxone_stats_set_stat_*` functions), and then you specify a number of details about what leaderboard information you want to retrieve.

Note that you can only retrieve a social leaderboard for int or real stats, but not for string stats, and that if you flag the `favourites_only` argument as `true`, then the results will only contain data for those friends that are marked by the user as "favorites".

IMPORTANT The user ID passed into this function should first be added to the statistics manager using the **xboxone_stats_add_user** function.

TIP Stats used in social leaderboards do not need to be registered as "Featured Stats" in the Partner Center.

This is an asynchronous function that will trigger the **Async Social** event when the task is finished.

Syntax:

```
xboxone_stats_get_social_leaderboard(user_id, stat, num_entries, start_rank,
start_at_user, ascending, favourites);
```

Argument	Type	Description
user_id	pointer	The user ID pointer.
stat	string	The stat to create the global leaderboard from.
num_entries	real	The number of entries from the leaderboard to retrieve.
start_rank	real	The rank in the leaderboard to start from (set to `0` if `start_at_user` is set to true.
start_at_user	boolean	Set to `true` to start at the user ID.
ascending	boolean	Set to `true` or ascending or `false` for descending order.
favourites_only	boolean	Set to `true` to show only friends that are marked as favourites.

Returns:

```
N/A
```

Triggers:

```
Asynchronous Social Event
```

Key	Type	Description
id	constant	The constant `achievement_leaderboard_info`
event	string	The string `"GetLeaderboardComplete"`.
userid	pointer	The user ID associated with the request.
error	real	0 if successful, some other value on error.
errorMessage	string	A string with an error message **[OPTIONAL]**
displayName	string	The unique ID for the leaderboard as defined on the provider dashboard.
numentries	real	The number of entries in the leaderboard that you have received.
PlayerN	string	The name of the player, where **N** is position within the received entries list.
PlayeridN	pointer	The unique user id of the player **N**
RankN	real	The rank of the player **N** within the leaderboard.
ScoreN	real	The score of the player **N**

Example:

```
xboxone_stats_get_social_leaderboard(user_id, "GlobalTime", 20, 1, false, true,
true);
```

In the code above we are querying a leaderboard for the stat `"GlobalTime"` with the first 20 entries starting on rank 1 in ascending order, selecting only friends marked as favourites.

We can catch the triggered callback using the the **Async Social** event.

```
if (async_load[? "id"] == achievement_leaderboard_info)
{
    if (async_load[? "event"] == "GetLeaderboardComplete")
    {
        global.numentries = async_load[? "numentries"];
        for (var i = 0; i < numentries; i++)
        {
            global.playername[i] = async_load[? "Player" + string(i)];
            global.playerid[i] = async_load[? "Playerid" + string(i)];
            global.playerrank[i] = async_load[? "Rank" + string(i)];
            global.playerscore[i] = async_load[? "Score" + string(i)];
        }
    }
}
```

The code above matches the response against the correct **id** and **event**, and store the information of the top favourite friends player names, ids, ranks and scores in global arrays.

xboxone_stats_get_stat

This function can be used to retrieve a single stat value from the stat manager for the given user. You supply the `user_id` as returned by the function **xboxone_get_user**, and then the `stat_name` as defined when you created it using the one of the `xboxone_stats_set_stat_*` functions. The return value can be either a string or a real (depending on the stat being checked) or the GML constant `undefined` if the stat does not exist or there has been an error.

IMPORTANT The user ID passed into this function should first be added to the statistics manager using the **xboxone_stats_add_user** function.

Syntax:

```
xboxone_stats_get_stat(user_id, stat_name)
```

Argument	Type	Description
user_id	pointer	The user ID pointer.
stat_name	string	The statistic to get.

Returns:

```
Real/String (The value for the given stat) or undefined
```

Example:

```
if (game_over == true)
{
    if (xboxone_stats_get_stat(user_id, "PercentDone") < 100)
    {
        var _val = (global.LevelsFinished / global.LevelsTotal)*100;
        xboxone_stats_set_stat_real(user_id, "PercentDone", _val);
    }
}
```

In the code above we are checking if the given user complete percentage (`"PercentDone"`) is less then 100 and if so updated it to a new value (using the xboxone_stats_set_stat_real function).

xboxone_stats_get_stat_names

This function can be used to retrieve all the defined stats from the stat manager for the given user. You supply the user ID as returned by the function xboxone_get_user, and the function returns an array of strings containing the statistics for the user. If an error occurs or the user has no stats, an empty array will be returned.

IMPORTANT The user ID passed into this function should first be added to the statistics manager using the xboxone_stats_add_user function.

Syntax:

```
xboxone_stats_get_stat_names(user_id)
```

Argument	Type	Description
user_id	pointer	The user ID pointer.

Returns:

```
Array (Array with all the stat names, strings)
```

Example:

```
var _stats = xboxone_stats_get_stat_names(user_id);
for (var i = 0; i < array_length(_stats); i++)
{
    xboxone_stats_delete_stat(user_id, _stats[i]);
}
```

In the code above we are getting an array of stat names (`_stats`) for a given user and looping though it deleting each stat (using the xboxone_stats_delete_stat function).

xboxone_stats_remove_user

This function can be used to remove (unregister) the given user from the statistics manager, performing a flush of the stat data to the live server. According to the Xbox documentation the game does not have to remove the user from the stats manager, as the GDK Extension will periodically flush the stats anyway.

To use the function, you supply the `user_id` as returned by the function xboxone_get_user, and the function will will return -1 if there was an error or the user ID was invalid, or any other value if the function was successfully called.

IMPORTANT The user ID passed into this function should first be added to the statistics manager using the xboxone_stats_add_user function.

IMPORTANT Removing the user can return an error if the statistics that have been set on the user are invalid (such as the stat names containing non-alphanumeric characters).

TIP If you want to flush the stats data to the live server at any time without removing the user, you can use the function xboxone_stats_flush_user.

This is an asynchronous function that will trigger the **Async Social** event when the task is finished.

Syntax:

```
xboxone_stats_remove_user(user_id);
```

Argument	Type	Description
user_id	pointer	The user ID pointer.

Returns:

```
Real (-1 on error, any other value otherwise)
```

Triggers:

```
Asynchronous Social Event
```

Key	Type	Description
id	constant	The constant `achievement_stat_event`
event	string	The string `"LocalUserRemoved"`.
userid	pointer	The user ID associated with the request.
error	real	0 if successful, some other value on error.
errorMessage	string	A string with an error message **[OPTIONAL]**

Example:

```
for(var i = 0; i < array_length(user_ids); i++)
{
    xboxone_stats_remove_user(user_ids[i]);
}
```

In the code above we are looping though an array of user ids (`user_ids`) and removing them from the statistics manager.

We can catch the triggered callbacks using the the Async Social event.

```
if (async_load[? "event"] == "LocalUserRemoved")
{
    if (async_load[? "error"] != 0)
    {
        show_debug_message("Error while removing user from statistics manager");
    }
}
```

The code above matches the response against the correct **event**, providing a failure message if **error** is not 0.

xboxone_stats_set_stat_int

This function can be used to set the value of a stat for the given user ID. You supply the user ID as returned by the function xboxone_get_user, then the stat string to set (if the stat string does not already exist then a new stat will be created and set to the given value) and a value (an **integer**) to set the stat to. Note that the stat name must be alphanumeric only, with no symbols or spaces.

IMPORTANT The user ID passed into this function should first be added to the statistics manager using the xboxone_stats_add_user function.

TIP When setting a stat value, any previous values will be overwritten, therefore it is up to you to determine if the stat value should be updated or not (ie. check that the high score is actually the highest) by comparing to the current stat value with the new one before setting it.

Syntax:

```
xboxone_stats_set_stat_int(user_id, stat_name, stat_value)
```

Argument	Type	Description
user_id	pointer	The user ID pointer.
stat_name	string	The statistic to set.
stat_value	integer	The value to set the stat to.

Returns:

```
Real (-1 on error, any other value otherwise)
```

Example:

```
var _highscore = 123;
xboxone_stats_set_stat_int(users_ids[0], "Highscore", _highscore);
```

In the code above we are setting the stat `"Highscore"` of the selected user (`user_ids[0]`) to a new value, overwriting any previous recorded value.

xboxone_stats_set_stat_real

This function can be used to set the value of a stat for the given user ID. You supply the user ID as returned by the function xboxone_get_user, then the stat string to set (if the stat string does not already exist then a new stat will be created and set to the given value) and a value (a **real**) to set the stat to. Note that the stat name must be alphanumeric only, with no symbols or spaces.

IMPORTANT The user ID passed into this function should first be added to the statistics manager using the xboxone_stats_add_user function.

TIP When setting the stat value, any previous values will be overwritten, therefore it is up to you to determine if the stat value should be updated or not (ie. check that the high score is actually the highest) by comparing to the current stat value with the new one before setting it.

Syntax:

```
xboxone_stats_set_stat_real(user_id, stat_name, stat_value)
```

Argument	Type	Description
user_id	pointer	The user ID pointer.
stat_name	string	The statistic to set.
stat_value	real	The value to set the stat to.

Returns:

```
Real (-1 on error, any other value otherwise)
```

Example:

```
var _bestTime = 123.45;
xboxone_stats_set_stat_real(users_ids[0], "BestTime", _bestTime);
```

In the code above we are setting the stat `"BestTime"` of the selected user (`user_ids[0]`) to a new value, overwriting any previous recorded value.

xboxone_stats_set_stat_string

This function can be used to set the value of a stat for the given user ID. You supply the user ID as returned by the function **xboxone_get_user**, then the stat string to set (if the stat string does not already exist then a new stat will be created and set to the given value) and a value (a **string**) to set the stat to. Note that the stat name must be alphanumeric only, with no symbols or spaces.

IMPORTANT The user ID passed into this function should first be added to the statistics manager using the **xboxone_stats_add_user** function.

TIP When setting the stat value, any previous values will be overwritten, therefore it is up to you to determine if the stat value should be updated or not (ie. check that the high score is actually the highest) by comparing to the current stat value with the new one before setting it.

Syntax:

```
xboxone_stats_set_stat_string(user_id, stat_name, stat_value)
```

Argument	Type	Description
user_id	pointer	The user ID pointer.
stat_name	string	The statistic to set.
stat_value	string	The value to set the stat to.

Returns:

```
Real (-1 on error, any other value otherwise)
```

Example:

```
xboxone_stats_set_stat_string(users_ids[0], "Team", "YoYo Games");
```

In the code above we are setting the stat `"Team"` of the selected user (`user_ids[0]`) to a new value, overwriting any previous recorded value.

In-App Purchases Module

A Microsoft Store game can have several In-App Purchases. Inside the Partner Center these are referred to as **add-ons** and can be of multiple types (consumables, durables, subscriptions, DLCs). For detailed instructions on how to configure those, check the Add-ons section on the Microsoft Partner Center Helpdesk page.

Contents of this module can be divided into the following sections:

- Product Functions
- Package Functions (DLCs)
- Structs (returned from some functions above)
- Constants

Functions

For helping the development process this API comes with a set of functions that will deal with all IAP (or add-on) needs. The available functions can be sub-dived into two main groups: the Product Functions that should be used with all the **non-package products** and the Package Functions that should be used with all the **package products**.

Product Functions

The following functions are provided for working with products:

- ms_iap_AcquireLicenseForDurables
- ms_iap_ReleaseLicenseForDurables
- ms_iap_CanAcquireLicenseForStoreId
- ms_iap_QueryAddOnLicenses
- ms_iap_QueryAssociatedProducts
- ms_iap_QueryConsumableBalanceRemaining
- ms_iap_QueryEntitledProducts
- ms_iap_QueryGameLicense
- ms_iap_QueryProductForCurrentGame
- ms_iap_QueryProductForPackage
- ms_iap_QueryProducts
- ms_iap_ReportConsumableFulfillment
- ms_iap_ShowAssociatedProductsUI
- ms_iap_ShowProductPageUI
- ms_iap_ShowPurchaseUI
- ms_iap_ShowRateAndReviewUI
- ms_iap_ShowRedeemTokenUI

Package Functions

The following functions are provided for working with packages (DLCs):

- ms_iap_AcquireLicenseForPackage
- ms_iap_ReleaseLicenseForPackage
- ms_iap_CanAcquireLicenseForPackage
- ms_iap_DownloadAndInstallPackages
- ms_iap_EnumeratePackages
- ms_iap_MountPackage
- ms_iap_UnmountPackage

ms_iap_AcquireLicenseForDurables

This function acquires a license for the current game's specified durable that the user is entitled to. This is only applicable to **Durable without package** add-on types. For **Durable with packages** (DLC) type, use ms_iap_AcquireLicenseForPackage instead.

This is an asynchronous function that will trigger the **Async In-App Purchase** event when the task is finished.

Syntax:

```
ms_iap_AcquireLicenseForDurables(user_id, store_id);
```

Argument	Type	Description
user_id	pointer	The user ID to use in the store context.
store_id	string	The store ID of the product.

Returns:

```
Real (In-App Purchase Request ID)
```

Triggers:

```
Asynchronous In-App Purchase Event
```

Key	Type	Description
type	string	The constant `"ms_iap_AcquireLicenseForDurables_result"`
id	real	The asynchronous request ID.
status	bool	Whether or not the asynchronous request succeeded.
store_id	string	The store ID used when acquiring the license.

Example:

```
var _userId = xboxone_get_activating_user();
var _storeId = global.products[0].storeId;

requestId = ms_iap_AcquireLicenseForDurables(_userId, _storeId);
```

In the code above first we get the user ID (`_userId`) of the activating user and we get the store ID (`_storeId`) of the product we want to acquire the license to (note that for convenience we keep the products stored in a global variable). The function call will then return a request ID (`requestId`) that can be used inside an Async In-App Purchase event.

```
if (async_load[? "type"] == "ms_iap_AcquireLicenseForDurables_result")
{
    if (async_load[? "id"] == requestId)
    {
        if (async_load[? "status"])
        {
            show_debug_message("License successfully acquired!");
        }
    }
}
```

The code above matches the response against the correct event **type** and **id**, providing a success message if **status** is true.

ms_iap_AcquireLicenseForPackage

Acquires a license for the current game's specified DLC that the user is entitled to use. This is only applicable to **Durable with package** add-on type. For **Durable** type, use ms_iap_AcquireLicenseForDurables instead.

This is an asynchronous function that will trigger the **Async In-App Purchase** event when the task is finished.

Syntax:

```
ms_iap_AcquireLicenseForPackage(user_id, package_id);
```

Argument	Type	Description
user_id	pointer	The user ID to use in the store context.
package_id	string	A string that uniquely identifies a store package.

Returns:

```
Real (In-App Purchase Request ID)
```

Triggers:

```
Asynchronous In-App Purchase Event
```

Key	Type	Description
type	string	The constant `"ms_iap_AcquireLicenseForPackage_result"`
id	real	The asynchronous request ID.
status	bool	Whether or not the asynchronous request succeeded.
package_id	string	The package ID used when acquiring the license.

Example:

```
var _userId = xboxone_get_activating_user();
var _packageId = global.packages[0].packageId;

requestId = ms_iap_AcquireLicenseForPackage(_userId, _storeId);
```

In the code above first we get the user ID (`_userId`) of the activating user and we get the package ID (`_packageId`) of the product we want to acquire the license to (note that for convenience we keep the products stored in a global variable). The function call will then return a request ID (`requestId`) that can be used inside an Async In-App Purchase event.

```
if (async_load[? "type"] == "ms_iap_AcquireLicenseForPackage_result")
{
    if (async_load[? "id"] == requestId)
    {
        if (async_load[? "status"])
        {
            show_debug_message("License for package successfully acquired!");
        }
    }
}
```

The code above matches the response against the correct event **type** and **id**, providing a success message if **status** is true.

ms_iap_CanAcquireLicenseForPackage

This function provides the ability for a game to determine if the user could acquire a license for a particular piece of content without actually acquiring that license and using up that user's concurrency slot. This cannot be used for the current base game (since it has a license already), but to determine if the user owns some DLC for that game.

This is an asynchronous function that will trigger the **Async In-App Purchase** event when it is finished.

Syntax:

```
ms_iap_CanAcquireLicenseForPackage(user_id, package_id);
```

Argument	Type	Description
user_id	pointer	The user ID to use in the store context.
package_id	string	A string that uniquely identifies a store package.

Returns:

```
Real (In-App Purchase Request ID)
```

Triggers:

```
Asynchronous In-App Purchase Event
```

Key	Type	Description
type	string	The string value `"ms_iap_CanAcquireLicenseForPackage_result"`.
async_id	real	The asynchronous request ID.
async_status	bool	Whether or not the asynchronous request succeeded.
package_id	string	A string that uniquely identifies a store package.

licensableSku	string	The SKU the user would be able to license.
licenseStatus	integer	Indicates if a user would be able to license a package: 0 - The package is not licensable to the user. 1 - The package is licensable to the user. 2 - The product is not individually licensable.

Example:

```
var _userId = xboxone_get_activating_user();
var _storeId = global.products[0].storeId;

requestId = ms_iap_AcquireLicenseForDurables(_userId, _storeId);
```

In the code above first we get the user ID (`_userId`) of the activating user and we get the store ID (`_storeId`) of the product we want to acquire the license to (note that for convenience we keep the products stored in a global variable). The function call will then return a request ID (`requestId`) that can be used inside an Async In-App Purchase event.

```
if (async_load[? "type"] == "ms_iap_CanAcquireLicenseForPackage_result")
{
    if (async_load[? "async_id"] == requestId)
    {
        if (async_load[? "licensableStatus"] == 1)
        {
            show_debug_message("The product is licensable!");
        }
    }
}
```

The code above matches the response against the correct event **type** and **async_id**, printing to the debug console if the current product is licensable to the provided user.

ms_iap_CanAcquireLicenseForStoreId

This function provides the ability for a game to determine if the user could acquire a license for a particular piece of content without actually acquiring that license and using up that user's concurrency slot. This is not used for the currently running game (since it has a license already), but to determine if the user owns other games from the same publisher, either to up-sell or to provide special benefits to loyal fans.

For example: this could be used by a racing game to determine whether to give bonus cars to users who owned prior versions of the game. This function is also used to determine whether or not the game should show content in its in game store based on whether or not the user already owns some content.

This is an asynchronous function that will trigger the **Async In-App Purchase** event when it is finished.

Syntax:

```
ms_iap_CanAcquireLicenseForStoreId(user_id, store_id);
```

Argument	Type	Description
user_id	pointer	The user ID to use in the store context.
store_id	string	The store ID of the product to check the license status.

Returns:

```
Real (In-App Purchase Request ID)
```

Triggers:

```
Asynchronous In-App Purchase Event
```

Key	Type	Description
type	string	The string value `"ms_iap_CanAcquireLicenseForStoreId_result"`.

async_id	real	The asynchronous request ID.
async_status	bool	Whether or not the asynchronous request succeeded.
store_id	string	The store ID of the product being checked.
licensableSku	string	The SKU the user would be able to license.
licenseStatus	integer	Indicates if a user would be able to license a package: 0 - The package is not licensable to the user. 1 - The package is licensable to the user. 2 - The product is not individually licensable.

Example:

```
var _userId = xboxone_get_activating_user();
var _storeId = global.products[0].storeId;

requestId = ms_iap_AcquireLicenseForDurables(_userId, _storeId);
```

In the code above first we get the user ID (`_userId`) of the activating user and we get the store ID (`_storeId`) of the product we want to acquire the license to (note that for convenience we keep the products stored in a global variable). The function call will then return a request ID (`requestId`) that can be used inside an Async In-App Purchase event.

```
if (async_load[? "type"] == "ms_iap_CanAcquireLicenseForStoreId_result")
{
    if (async_load[? "async_id"] == requestId)
    {
        if (async_load[? "licensableStatus"] == 1)
        {
            show_debug_message("The product is licensable!");
        }
    }
}
```

The code above matches the response against the correct event **type** and **async_id**, printing to the debug console if the current product is licensable to the provided user.

ms_iap_DownloadAndInstallPackages

Downloads and installs the specified store packages.

This is an asynchronous function that will trigger the **Async In-App Purchase** event when it is finished. The async completion message will only be raised when the package is actually **installed** (not just registered).

Syntax:

```
ms_iap_DownloadAndInstallPackages(user_id, package_ids);
```

Argument	Type	Description
user_id	pointer	The user ID to use in the store context.
package_ids	string[]	An array of strings that uniquely identify the store packages.

Returns:

```
Real (In-App Purchase Request ID)
```

Triggers:

```
Asynchronous In-App Purchase Event
```

Key	Type	Description
type	string	The string value `"ms_iap_DownloadAndInstallPackages_result"`.
id	real	The asynchronous request ID.
status	bool	Whether or not the asynchronous request succeeded.
package_ids	string[]	An array of strings that uniquely identify the store packages.

Example:

```
var _userId = xboxone_get_activating_user();
var _package1 = global.package[0].packageId;
var _package2 = global.package[1].packageId;
var _package2 = global.package[2].packageId;

requestId = ms_iap_AcquireLicenseForDurables(_userId, [_package1, _package2,
_package3]);
```

In the code above first we get the user ID (`_userId`) of the activating user and after that we execute the function passing in an array of packages to be downloaded and installed. The function call will then return a request ID (`requestId`) that can be used inside an **Async In-App Purchase** event.

```
if (async_load[? "type"] == "ms_iap_DownloadAndInstallPackages_result")
{
    if (async_load[? "id"] == requestId)
    {
        if (async_load[? "status"] == 1)
        {
            show_debug_message(async_load[? "package_ids"]);
        }
    }
}
```

The code above matches the response against the correct event **type** and **id**, printing to the debug console an array will all the installed packages IDs.

ms_iap_EnumeratePackages

This functions enumerates the results of a package query.

This is an asynchronous function that will trigger the **Async In-App Purchase** event when the task is finished.

Syntax:

```
ms_iap_EnumeratePackages(package_kind, scope);
```

Argument	Type	Description
package_kind	constant	The value that indicates whether to enumerate app packages or content packages (see Package Kinds).
scope	constant	The scope of the installation packages (see Package Scopes).

Returns:

```
Real (In-App Purchase Request ID)
```

Triggers:

```
Asynchronous In-App Purchase Event
```

Key	Type	Description
type	string	The string value `"ms_iap_EnumeratePackages_result"`.
id	real	The asynchronous request ID.
status	bool	Whether or not the asynchronous request succeeded.
results	array	An array of Package Details structs.

Example:

```
requestId = ms_iap_EnumeratePackages(e_ms_iap_PackageKind_Content,
e_ms_iap_PackageEnumerationScope_ThisOnly);
```

In the code above first we request an enumeration all the packages of type content (`e_ms_iap_PackageKind_Content`) and whose scope is limited to the current game (`e_ms_iap_PackageEnumerationScope_ThisOnly`). The function call will then return a request ID (`requestId`) that can be used inside an Async In-App Purchase event.

```
if (async_load[? "type"] == "ms_iap_EnumeratePackages_result")
{
    if (async_load[? "id"] == requestId)
    {
        if (async_load[? "status"] == 1)
        {
            show_debug_message(async_load[? "results"]);
        }
    }
}
```

The code above matches the response against the correct event **type** and **id**, printing to the debug console an array will all the Package Details structs.

ms_iap_MountPackage

This function mounts the installation of specified content.

This is an asynchronous function that will trigger the **Async In-App Purchase** event when it is finished.

Syntax:

```
ms_iap_MountPackage(package_id);
```

Argument	Type	Description
package_id	string	A string that uniquely identifies the installed package on the disk. Pass in the **packageIdentifier** field from the Package Details struct returned from the ms_iap_EnumeratePackages async callback.

Returns:

```
Real (In-App Purchase Request ID)
```

Triggers:

```
Asynchronous In-App Purchase Event
```

Key	Type	Description
type	string	The string value `"ms_iap_MountPackage_result"`.
id	real	The asynchronous request ID.
package_id	string	A string that uniquely identifies the installed package on the disk.
mount_path	string	The path to the mounted installation.

Example:

```
var _package = global.package[0].packageId;

requestId = ms_iap_MountPackage(_package);
```

In the code above first we get the user ID (`_userId`) of the activating user and after that we execute the function passing in an array of packages to be downloaded and installed. The function call will then return a request ID (`requestId`) that can be used inside an **Async In-App Purchase** event.

```
if (async_load[? "type"] == "ms_iap_MountPackage_result")
{
    if (async_load[? "id"] == requestId)
    {
        show_debug_message("Package ID: " + async_load[? "package_id"]);
        show_debug_message("Mount Path: " + async_load[? "mount_path"]);
    }
}
```

The code above matches the response against the correct event **type** and **id**, printing to the debug console the current packages ID and it's mount path on disk.

ms_iap_QueryAddOnLicenses

Retrieves the licenses the user was granted for Add-ons (also known as a durable without bits) of the currently running game. It is generally recommended to use dlc, rather than add-ons, but this API exists for the few that choose to use add-ons anyways.

Add-ons are typically content or features that require a purchase to unlock but don't require a download because they are built into the game. They do not work for games that have discs, or may ever have discs.

This is an asynchronous function that will trigger the **Async In-App Purchase** event when it is finished.

Syntax:

```
ms_iap_QueryAddOnLicenses(user_id);
```

Argument	Type	Description
user_id	pointer	The user ID to use in the store context.

Returns:

```
Real (In-App Purchase Request ID)
```

Triggers:

```
Asynchronous In-App Purchase Event
```

Key	Type	Description
type	string	The constant `"ms_iap_QueryAddOnLicenses_result"`
id	real	The asynchronous request ID.
status	bool	Whether or not the asynchronous request succeeded.
result	array	An array of Addon License Details structs.

Example:

```
var _userId = xboxone_get_activating_user();

requestId = ms_iap_QueryAddOnLicenses(_userId);
```

In the code above first we get the user ID (`_userId`) of the activating user and we call the function with it. The function call will then return a request ID (`requestId`) that can be used inside an Async In-App Purchase event.

```
if (async_load[? "type"] == "ms_iap_QueryAddOnLicenses_result")
{
    if (async_load[? "id"] == requestId)
    {
        if (async_load[? "status"])
        {
            show_debug_message(async_load[? "result"]);
        }
    }
}
```

The code above matches the response against the correct event **type** and **id**, printing to the debug console if the result of the query; an array of Addon License Details structs.

ms_iap_QueryAssociatedProducts

Gets store listing information for the products that can be purchased from within the current game. This API will only return products that can be browsed within the store including any Add-on products that do not have a store page as long as they are not expired. Any product that is hidden, expired, or taken down from the store will not be returned in the results from this API. If you need store info from a hidden, expired, or taken down product use the ms_iap_QueryProducts passing in the product's StoreID.

This is an asynchronous function that will trigger the **Async In-App Purchase** event when it is finished.

Syntax:

```
ms_iap_QueryAssociatedProducts(user_id, product_kinds);
```

Argument	Type	Description
user_id	pointer	The user ID to use in the store context.
product_kinds	constant	The type of products to return. For more information read the Product Kinds section.

Returns:

```
Real (In-App Purchase Request ID)
```

Triggers:

```
Asynchronous In-App Purchase Event
```

Key	Type	Description
type	string	The constant `"ms_iap_AcquireLicenseForDurables_result"`
id	real	The asynchronous request ID.
status	bool	Whether or not the asynchronous request succeeded.

results	array	An array of **Product Details** structs.

Example:

```
var _userId = xboxone_get_activating_user();
var _product_kinds = e_ms_iap_ProductKind_Consumable;

requestId = ms_iap_QueryAssociatedProducts(_userId, _product_kinds);
```

In the code above first we get the user ID (`_userId`) of the activating user, afterwards we create a filter for consumable products (`_product_kinds`) and we call the function with both arguments. The function call will then return a request ID (`requestId`) that can be used inside an **Async In-App Purchase** event.

```
if (async_load[? "type"] == "ms_iap_QueryAssociatedProducts_result")
{
    if (async_load[? "id"] == requestId)
    {
        if (async_load[? "status"])
        {
            show_debug_message(async_load[? "result"]);
        }
    }
}
```

The code above matches the response against the correct event **type** and **id**, printing to the debug console the result of the query; an array of **Product Details** structs.

ms_iap_QueryConsumableBalanceRemaining

This function gets the consumable balance remaining for the specified product ID.

This is an asynchronous function that will trigger the **Async In-App Purchase** event when it is finished.

Syntax:

```
ms_iap_QueryConsumableBalanceRemaining(user_id, store_id);
```

Argument	Type	Description
user_id	pointer	The user ID to use in the store context.
store_id	string	The ID of the consumable to retrieve the balance for.

Returns:

```
Real (In-App Purchase Request ID)
```

Triggers:

```
Asynchronous In-App Purchase Event
```

Key	Type	Description
type	string	The constant `"ms_iap_QueryConsumableBalanceRemaining_result"`
id	real	The asynchronous request ID.
status	bool	Whether or not the asynchronous request succeeded.
store_id	string	The ID of the consumable to retrieve the balance for.
quantity	real	The remaining quantity of the consumable.

Example:

```
var _userId = xboxone_get_activating_user();
var _storeId = global.products[3].storeId;

requestId = ms_iap_QueryConsumableBalanceRemaining(_userId, _storeId);
```

In the code above first we get the user ID (`_userId`) of the activating user and we get the store ID (`_storeId`) of the product we want to query the balance for (note that for convenience we keep the products stored in a global variable). The function call will then return a request ID (`requestId`) that can be used inside an Async In-App Purchase event.

```
if (async_load[? "type"] == "ms_iap_QueryConsumableBalanceRemaining_result")
{
    if (async_load[? "id"] == requestId)
    {
        if (async_load[? "status"])
        {
            var _quantity = async_load[? "quantity"];
            show_debug_message("Amount of potions left: " + string(_quantity));
        }
    }
}
```

The code above matches the response against the correct event **type** and **id**, providing a success message with the current available quantity given that the **status** value is true.

ms_iap_QueryEntitledProducts

This function provides the store product information for all add-ons, dlc, and consumables related to the current game that the user has an entitlement to.

This is an asynchronous function that will trigger the **Async In-App Purchase** event when it is finished.

Syntax:

```
ms_iap_QueryEntitledProducts(user_id, product_kinds);
```

Argument	Type	Description
user_id	pointer	The user ID to use in the store context.
product_kinds	constant	The types of products to return. For more information read the Product Kinds section.

Returns:

```
Real (In-App Purchase Request ID)
```

Triggers:

```
Asynchronous In-App Purchase Event
```

Key	Type	Description
type	string	The constant `"ms_iap_QueryEntitledProducts_result"`
id	pointer	The asynchronous request ID.
status	bool	Whether or not the asynchronous request succeeded.
results	array	An array of Product Details structs.

Example:

```
var _userId = xboxone_get_activating_user();
var _product_kinds = e_ms_iap_ProductKind_Durable;

requestId = ms_iap_QueryEntitledProducts(_userId, _product_kinds);
```

In the code above first we get the user ID (`_userId`) of the activating user, afterwards we create a filter for durable products (`_product_kinds`) and we call the function with both arguments. The function call will then return a request ID (`requestId`) that can be used inside an Async In-App Purchase event.

```
if (async_load[? "type"] == "ms_iap_QueryEntitledProducts_result")
{
    if (async_load[? "id"] == requestId)
    {
        if (async_load[? "status"])
        {
            show_debug_message(async_load[? "result"]);
        }
    }
}
```

The code above matches the response against the correct event **type** and **id**, printing to the debug console the result of the query; an array of Product Details structs.

ms_iap_QueryGameLicense

This function retrieves information about the license that was acquired to allow the app to launch.

This is an asynchronous function that will trigger the **Async In-App Purchase** event when it is finished.

Syntax:

```
ms_iap_QueryGameLicense(user_id);
```

Argument	Type	Description
user_id	pointer	The user ID to use in the store context.

Returns:

```
Real (In-App Purchase Request ID)
```

Triggers:

```
Asynchronous In-App Purchase Event
```

Key	Type	Description
type	string	The constant `"ms_iap_QueryGameLicense_result"`
id	pointer	The asynchronous request ID.
status	bool	Whether or not the asynchronous request succeeded.
expirationDate	real	Expiration date of the license.
isActive	bool	Indicates whether the license is active.
isTrial	bool	Indicates whether the license is a trial license.

isTrialOwnedByTheUser	bool	Indicates whether the trial is owned by the associated user. If on PC, this will be the currently signed in user to the Windows Store App.
isDiscLicense	bool	Indicates whether the license is a disc license.
skuStoreId	string	The store ID.
trialUniqueId	string	The unique ID for the trial.
trialTimeRemainingInSeconds	real	Amount of time remaining for the trial license.

Example:

```
var _userId = xboxone_get_activating_user();

requestId = ms_iap_QueryGameLicense(_userId);
```

In the code above first we get the user ID (`_userId`) of the activating user and we call the function with it. The function call will then return a request ID (`requestId`) that can be used inside an Async In-App Purchase event.

```
if (async_load[? "type"] == "ms_iap_QueryGameLicense_result")
{
    if (async_load[? "id"] == requestId)
    {
        if (async_load[? "status"])
        {
            show_debug_message(async_load[? "skuStoreId"]);
        }
    }
}
```

The code above matches the response against the correct event **type** and **id**, printing to the debug console the **skuStoreId** of the current game license.

ms_iap_QueryProductForCurrentGame

This function provides store product information for the currently running game, such as its SKUs, availabilities and other metadata.

This is an asynchronous function that will trigger the **Async In-App Purchase** event when it is finished.

Syntax:

```
ms_iap_QueryProductForCurrentGame(user_id);
```

Argument	Type	Description
user_id	pointer	The user ID to use in the store context.

Returns:

```
Real (In-App Purchase Request ID)
```

Triggers:

```
Asynchronous In-App Purchase Event
```

Key	Type	Description
type	string	The constant `"ms_iap_QueryProductForCurrentGame_result"`
id	real	The asynchronous request ID.
status	bool	Whether or not the asynchronous request succeeded.
results	array	An array of Product Details structs.

Example:

```
var _userId = xboxone_get_activating_user();

requestId = ms_iap_QueryProductForCurrentGame(_userId);
```

In the code above first we get the user ID (`_userId`) of the activating user and we call the function with it. The function call will then return a request ID (`requestId`) that can be used inside an Async In-App Purchase event.

```
if (async_load[? "type"] == "ms_iap_QueryProductForCurrentGame_result")
{
    if (async_load[? "async_id"] == requestId)
    {
        if (async_load[? "status"])
        {
            show_debug_message(async_load[? "result"]);
        }
    }
}
```

The code above matches the response against the correct event **type** and **id**, printing to the debug console the result of the query; an array of Product Details structs.

ms_iap_QueryProductForPackage

This function retrieves store product information for the specified package.

This is an asynchronous function that will trigger the **Async In-App Purchase** event when it is finished.

Syntax:

```
ms_iap_QueryProductForPackage(user_id, store_id, product_kinds);
```

Argument	Type	Description
user_id	pointer	The user ID to use in the store context.
store_id	string	A string that uniquely identifies a store package.
product_kinds	constant	The type of products to find. For more information read the **Product Kinds** section.

Returns:

```
Real (In-App Purchase Request ID)
```

Triggers:

```
Asynchronous In-App Purchase Event
```

Key	Type	Description
type	string	The constant `"ms_iap_QueryProductForPackage_result"`
id	real	The asynchronous request ID.
status	bool	Whether or not the asynchronous request succeeded.
results	array	An array of **Product Details** structs.

Example:

```
var _userId = xboxone_get_activating_user();
var _packageId = "XXXXXXXXXXX";
var _product_kinds = e_ms_iap_ProductKind_Consumable;

requestId = ms_iap_QueryProductForPackage(_userId, _packageId, _product_kinds);
```

In the code above first we get the user ID (`_userId`) of the activating user, we reference the package ID (`_packageId`) of the target game and afterwards we create a filter for consumable products (`_product_kinds`) finally we call the function with all those arguments. The function call will then return a request ID (`requestId`) that can be used inside an Async In-App Purchase event.

```
if (async_load[? "type"] == "ms_iap_QueryProductForPackage_result")
{
    if (async_load[? "id"] == requestId)
    {
        if (async_load[? "status"])
        {
            show_debug_message(async_load[? "result"]);
        }
    }
}
```

The code above matches the response against the correct event **type** and **id**, printing to the debug console the result of the query; an array of Product Details structs.

ms_iap_QueryProducts

Returns listing information for the specified products that are associated with the current game, regardless of whether the products are currently available for purchase within the current game.

This is an asynchronous function that will trigger the **Async In-App Purchase** event when it is finished.

Syntax:

```
ms_iap_QueryProducts(user_id, product_kinds, store_ids, action_filters);
```

Argument	Type	Description
user_id	pointer	The user ID to use in the store context.
product_kinds	constant	The types of products to return. For more information read the **Product Kinds** section.
store_ids	string[]	Restricts the results to the given product IDs.
action_filters	string[]	Restricts the results by some action stored in the product document. By default, this API returns all products, even if they are not purchasable, but you can restrict this to *"Purchase"* if you only want purchasable, or *"License"* if you only want licensable. Other action filters include *"Fulfill"*, *"Browse"*, *"Curate"*, *"Details"*, and *"Redeem"*.

Returns:

```
Real (In-App Purchase Request ID)
```

Triggers:

```
Asynchronous In-App Purchase Event
```

Key	Type	Description
type	string	The constant `"ms_iap_QueryProducts_result"`
id	real	The asynchronous request ID.
status	bool	Whether or not the asynchronous request succeeded.
results	array	An array of Product Details structs.

Example:

```
var _userId = xboxone_get_activating_user();
var _product_kinds = e_ms_iap_ProductKind_Consumable;
var _action_filter = ["Purchase"];

requestId = ms_iap_QueryProducts(_userId, _product_kinds, 0, _action_filter );
```

In the code above first we get the user ID (`_userId`) of the activating user, afterwards we create a filter for consumable products (`_product_kinds`) and finally we call the function with all those arguments providing no store ID filter but selecting only products that are available for purchase. The function call will then return a request ID (`requestId`) that can be used inside an Async In-App Purchase event.

```
if (async_load[? "type"] == "ms_iap_QueryProducts_result")
{
    if (async_load[? "id"] == requestId)
    {
        if (async_load[? "status"])
        {
            show_debug_message(async_load[? "result"]);
        }
    }
}
```

The code above matches the response against the correct event **type** and **id**, printing to the debug console the result of the query; an array of Product Details structs.

ms_iap_ReleaseLicenseForDurables

This function will release a package license assigned to this console and it is intended to be be used with products of type **Durable**. For **Durable with package** type, use **ms_iap_ReleaseLicenseForPackage** instead.

Syntax:

```
ms_iap_ReleaseLicenseForDurables(store_id);
```

Argument	Type	Description
store_id	string	The store ID of the product.

Returns:

```
Real (-1 if there was an error, otherwise 0)
```

Example:

```
var _storeId = global.products[0].storeId;

var _error = ms_iap_ReleaseLicenseForDurables(_storeId);

if (_error != 0)
{
    show_debug_message("There was an error trying to release the license");
}
```

In the code above first we get the product ID (`_storeId`) conveniently stored inside a global array and call the function with this value. The returned value (`_error`) is then checked to see if the function was successful or not.

ms_iap_ReleaseLicenseForPackage

This function will release a package license assigned to this console and it is intended to be be used with products of type **Durable with package**. For **Durable** type, use **ms_iap_ReleaseLicenseForDurables** instead.

Syntax:

```
ms_iap_ReleaseLicenseForPackage(package_id);
```

Argument	Type	Description
package_id	string	The store ID of the product.

Returns:

```
Real (-1 if there was an error, otherwise 0)
```

Example:

```
var _packageId = global.packages[0].packageId;

var _error = ms_iap_ReleaseLicenseForPackage(_packageId);

if (_error != 0)
{
    show_debug_message("There was an error trying to release the license for the
package!");
}
```

In the code above first we get the package ID (`_packageId`) conveniently stored inside a global array and call the function with this value. The returned value (`_error`) is then checked to see if the function was successful or not.

ms_iap_ReportConsumableFulfillment

Consumes the specified quantity of a consumable. See Consumable based ecosystems for more information on implementing and using consumable products. This is an asynchronous function that will trigger the Asynchronous In-App Purchase Event when it is finished.

Syntax:

```
ms_iap_ReportConsumableFulfillment(user_id, store_id, quantity, tracking_id);
```

Argument	Type	Description
user_id	pointer	The user ID to use in the store context.
store_id	string	The Store ID of the consumable add-on that you want to report as fulfilled.
quantity	integer	The number of units of the consumable add-on that you want to report as fulfilled. For a Store-managed consumable (that is, a consumable where Microsoft keeps track of the balance), specify the number of units that have been consumed. For a game-managed consumable (that is, a consumable where the developer keeps track of the balance), specify 1.
tracking_id	string	A developer-supplied GUID that identifies the specific transaction that the fulfillment operation is associated with for tracking purposes.

Returns:

```
Real (In-App Purchase Request ID)
```

Triggers:

```
Asynchronous In-App Purchase Event
```

Key	Type	Description
type	string	The constant `"ms_iap_ReportConsumableFulfillment_result"`
id	real	The asynchronous request ID.
status	bool	Whether or not the asynchronous request succeeded.
store_id	string	The store ID of the product.
consumed_quantity	integer	The amount of product that was consumed.
available_quantity	integer	The amount of product that still remains in the user's possession.

Example:

```
var _userId = xboxone_get_activating_user();
var _storeId = global.products[0].storeId;
var _trackingId = guid_generate(); // This function generates a unique identifier for
tracking purposes.

requestId = ms_iap_ReportConsumableFulfillment(_userId, _storeId, 10, _trackingId);
```

In the code above first we get the user ID (`_userId`) of the activating user and we get the store ID (`_storeId`) of the product we want to report as consumed (note that for convenience we keep the products stored in a global variable) and finally we generate a tracking ID (`_trackingId`) required to perform the request. The function call will then return a request ID (`requestId`) that can be used inside an Async In-App Purchase event.

```
if (async_load[? "type"] == "ms_iap_ReportConsumableFulfillment_result")
{
    if (async_load[? "id"] == requestId)
    {
        if (async_load[? "status"])
        {
            show_debug_message("We consumed: " + string(async_load[?
"consumed_quantity"]));
            show_debug_message("We still have: " + string(async_load[?
"available_quantity"]));
        }
    }
}
```

The code above matches the response against the correct event **type** and **async_id**, printing to the debug console the current consumed and available quantities; upon a successful task.

ms_iap_ShowAssociatedProductsUI

This function will open up the Microsoft Store App and show the set of available add-ons associated with the game. This can be further filtered by product type.

Syntax:

```
ms_iap_ShowAssociatedProductsUI(user_id, store_id, product_kinds);
```

Argument	Type	Description
user_id	pointer	The user ID to use in the store context.
store_id	string	The store ID of the product.
product_kinds	constant	The type of products to show. For more information read the **Product Kinds** section.

Returns:

```
N/A
```

Example:

```
var _userId = xboxone_get_activating_user();
var _storeId = global.gameStoreId;
var _product_kinds = e_ms_iap_ProductKind_Consumable;

ms_iap_ShowAssociatedProductsUI(_userId, _storeId, _product_kinds);
```

In the code above first we get the user ID (`_userId`) of the activating user, we reference the store ID (`_storeId`) of current game (stored inside a global variable) and afterwards we create a filter for consumable products (`_product_kinds`) finally we call the function with all those arguments. The function call will show up the Microsoft Store App overlay with the associated consumable products.

ms_iap_ShowProductPageUI

This function will open up the Store app directly to the **Product Details Page** (PDP) of the provided product. This allows titles that have not integrated with the purchase flow or an in-game store UI to still drive users to products related to their title and the purchase flow found on the Product Details Page.

Syntax:

```
ms_iap_ShowProductPageUI(user_id, store_id);
```

Argument	Type	Description
user_id	pointer	The user ID to use in the store context.
store_id	string	The store ID of the product.

Returns:

```
N/A
```

Example:

```
var _userId = xboxone_get_activating_user();
var _storeId = global.products[0].storeId;

ms_iap_ShowProductPageUI(_userId, _storeId);
```

In the code above first we get the user ID (`_userId`) of the activating user, we reference the store ID (`_storeId`) of product we want to check and finally we call the function with both arguments. The function call will show up the Microsoft Store App overlay directly on the **Product Details Page** (PDP) of the provided Product ID.

ms_iap_ShowPurchaseUI

This function begins the purchase UI overlay for the specified product.

Syntax:

```
ms_iap_ShowPurchaseUI(user_id, store_id, name, json);
```

Argument	Type	Description
user_id	pointer	The user ID to use in the store context.
store_id	string	The store ID of the product.
name	string	Name of the product to purchase.
json	string	A JSON blob (JSON formatted string) that is handed to the purchase flow. Allows for insertion of custom campaign IDs, so you can track how the purchase started.

Returns:

```
N/A
```

Example:

```
var _userId = xboxone_get_activating_user();
var _storeId = global.products[0].storeId;

ms_iap_ShowPurchaseUI(_userId, _storeId, undefined, undefined);
```

In the code above first we get the user ID (`_userId`) of the activating user, we reference the store ID (`_storeId`) of product we want to purchase and finally we call the function with both arguments (providing no name/json, they are not demanding). The function call will show up the purchase UI overlay for the specified product.

ms_iap_ShowRateAndReviewUI

Displays a system dialog to pop up to allow the user to provide a review for the current game or decline to do so.

Note: If the system detects a game is calling this excessively, it will hide the dialog.

Syntax:

```
ms_iap_ShowRateAndReviewUI(user_id);
```

Argument	Type	Description
user_id	pointer	The user ID to use in the store context.

Returns:

```
N/A
```

Example:

```
var _userId = xboxone_get_activating_user();

ms_iap_ShowRateAndReviewUI(_userId);
```

In the code above first we get the user ID (`_userId`) of the activating user and finally we call the function with it's value. The function call will show a system dialog pop up to allow the user to provide a review for the current game.

ms_iap_ShowRedeemTokenUI

This function triggers a token redemption for a given user and specified token.

Syntax:

```
ms_iap_ShowRedeemTokenUI(user_id, token, allowed_store_ids, disallow_cvs_redemption);
```

Argument	Type	Description
user_id	pointer	The user ID to use in the store context.
token	string	The token to redeem. This value cannot be undefined, if you want to bring up the UI without providing a code to pre-populate in the UI pass in a **single space**.
allowed_store_ids	string[]	An array of product store IDs. This allows you to restrict the 5x5 codes to only work with specific products.
disallow_cvs_redemption	bool	Prevents CSV (giftcard/ money style 5x5s) from being redeemed.

Returns:

```
N/A
```

Example:

```
var _userId = xboxone_get_activating_user();
var _token = global.finalWeaponToken;

ms_iap_ShowRateAndReviewUI(_userId, _token, undefined, false);
```

In the code above first we get the user ID (`_userId`) of the activating user and reference a token to be redeemed (`_token`), finally we call the function with those values, applying no restrictions whatsoever. The function call will triggers a token redemption for a given user and specified token.

ms_iap_UnmountPackage

This function unmounts the installation of specified content.

This is an asynchronous function that will trigger the **Async In-App Purchase** event when it is finished.

Syntax:

```
ms_iap_UnmountPackage(package_id);
```

Argument	Type	Description
package_id	string	A string that uniquely identifies the installed package on the disk. Pass in the **packageIdentifier** field from the Package Details struct returned from the ms_iap_EnumeratePackages async callback.

Returns:

```
Real (In-App Purchase Request ID)
```

Triggers:

```
Asynchronous In-App Purchase Event
```

Key	Type	Description
type	string	The string value `"ms_iap_UnmountPackage_result"`.
id	real	The asynchronous request ID.
package_id	string	A string that uniquely identifies the installed package on the disk.
mount_path	string	The path to the unmounted installation.

Example:

```
var _package = global.package[0].packageId;

requestId = ms_iap_UnmountPackage(_package);
```

In the code above first we get the user ID (`_userId`) of the activating user and after that we execute the function passing in an array of packages to be downloaded and installed. The function call will then return a request ID (`requestId`) that can be used inside an **Async In-App Purchase** event.

```
if (async_load[? "type"] == "ms_iap_UnmountPackage_result")
{
    if (async_load[? "id"] == requestId)
    {
        show_debug_message("Package ID: " + async_load[? "package_id"]);
        show_debug_message("Finished unmount!");
    }
}
```

The code above matches the response against the correct event **type** and **id**, printing to the debug console the current packages ID and a success message.

Structs

Some of the API asynchronous callback responses return data in the form of structs. This sections aims to deliver detailed information on each of the structs used within the In-App Purchases Module context.

The following **structs** (structures) are used as return members of API function calls:

- Addon License Details
- Package Details
- Product Details

Addon License Details

This struct is returned as an async result of the call to ms_iap_QueryAddOnLicenses and it contains details about an add-on license.

Property	Type	Description
skuStoreId	string	The SKU ID for the license.
isActive	bool	Indicates if the license is active.
expirationDate	real	Expiration date of the license (**-1**, if the license doesn't expire)
inAppOfferToken	string	The title defined offer token that you can use to map items internally. For example: *"com.company.product.itemname"*.

The entity described above is a **struct** meaning it's properties can be accessed using the **dot** operator much like when accessing instance variables, for example:

```
var _addonLicenseDetails = global.addonLicenses[0];

show_debug_message(_packageDetails.packageIdentifier);
show_debug_message(_packageDetails.version);
```

The code above will grab a package detail struct from a global variable (where they were previously store for this sample) and we are accessing the **packageIdentifier** and **version** properties of that struct.

Package Details

This struct is returned as an async result of the call to ms_iap_EnumeratePackages and it contains details about an installation.

Property	Type	Description
packageIdentifier	string	A string that uniquely identifies the installed package on the disk.
version	string	A store managed consumable product.
kind	constant	The value that indicates whether the package is an app package or a content package (see Package Kinds)
displayName	string	The display name.
description	string	The description of the package.
publisher	string	The publisher of the package.
storeId	string	The unique ID of the product.
installing	bool	The Boolean that indicates whether the package is currently installing.

The entity described above is a **struct** meaning it's properties can be accessed using the **dot** operator much like when accessing instance variables, for example:

```
var _packageDetails = global.packageDetails[0];

show_debug_message(_packageDetails.packageIdentifier);
show_debug_message(_packageDetails.version);
```

The code above will grab a package detail struct from a global variable (where they were previously store for this sample) and we are accessing the **packageIdentifier** and **version** properties of that struct.

Product Details

This struct is returned as an async result of the call to the following API function calls:

- ms_iap_QueryAssociatedProducts
- ms_iap_QueryEntitledProducts
- ms_iap_QueryProductForCurrentGame
- ms_iap_QueryProductForPackage
- ms_iap_QueryProducts

and it contains details that describe a store product.

Property	Type	Description
storeId	string	The product ID.
title	string	The title of the product.
description	string	A description of the store product.
language	string	The International Organization of Standards (ISO) identifier representing the language the title and description strings are (**more details**)
inAppOfferToken	string	Game defined offer token that you can use to map items internally. For example: "*com.company.product.itemname*".
linkUri	string	The URI to the product.
productKind	constant	Indicates the type of store product. For more information read the **Product Kinds** section.
price	struct	The price information for the store product (read **Price** below)
hasDigitalDownload	bool	Indicates whether the store product has a digital download.
isInUserCollection	bool	Indicates if the product is in the user collection.

keywords	string[]	Keywords associated with the store product.
images	array	Array of images associated with the product (read **Image** below)

Price

Price details inside the product struct use their own **struct** with data, following the schema below:

Property	Type	Description
basePrice	real	The normal non-promotional price or MSRP of the product.
price	real	The actual price that the user would pay if they purchased the item.
recurrecePrice	real	The recurrence price.
currencyCode	string	The currency code for the price.
formattedBasePrice	string	The formatted basePrice that can be shown in the game's UI.
formattedPrice	string	The formatted price that should be used in your UI to advertise the product.
formattedRecurrentPrice	string	The formatted recurrence price.
isOnSale	bool	Indicates whether the product is on sale.
saleEndData	real	The end date for the sale.

Image

Image details inside the product struct use their own array of **structs** with data, following the schema below:

Property	Type	Description
uri	real	The URI to the image.
height	real	The height of the image.
width	real	The width of the image.

caption	string	The caption for the image.
imagePurposeTag	string	A string containing a tag indicating the purpose of the image.

The entity described above is a **struct** meaning it's properties can be accessed using the **dot** operator much like when accessing instance variables, for example:

```
var _productDetails = global.productDetails[0];

show_debug_message(_productDetails.description);
show_debug_message(_productDetails.price.basePrice);
```

The code above will grab a product detail struct from a global variable (where they were previously store for this sample) and we are accessing the **description** text and **basePrice** (inside the price struct) properties of that struct.

Constants

Some of the API function calls and asynchronous callback responses use constants. This sections aims to deliver detailed information on each of the constants and its meaning within the In-App Purchases Module context.

The following constants (enumerations) are used as filter arguments and return values for some API queries:

- Package Kinds
- Package Scopes
- Product Kinds

Package Kinds

Package kind indicates the package type. They are used as a filter for the ms_iap_EnumeratePackages function to acquire information about packages of a certain type. It is also a member of the Package Details struct which describes a store product.

Constant	Description
`e_ms_iap_PackageKind_Game`	The installation package contains a game.
`e_ms_iap_PackageKind_Content`	The installation package contains downloadable content.

Package Scopes

Package scope indicates the scope of packages to be returned when installation packages are being enumerated while using the ms_iap_EnumeratePackages function call.

Constant	Description
`e_ms_iap_PackageEnumerationScope_ThisOnly`	Scope is limited to just apps or content associated with the calling process.
`e_ms_iap_PackageEnumerationScope_ThisAndRelated`	Scope includes apps or content associated with the calling process and also includes apps or content that are associated with any package the calling process has added to its RelatedProducts section of its game config file.

Products Kinds

Product kinds indicates the product type. They are used as a filter for many IAP queries to acquire information about products of a certain type. It is also a member of the Product Details struct which describes a store product. The product kind is represented as a flagged constant meaning it can be combined using the **bit-wise or operator** to represent multiple types of product at once.

Constant	Description
`e_ms_iap_ProductKind_None`	Not a product type.
`e_ms_iap_ProductKind_Consumable`	A store managed consumable product.
`e_ms_iap_ProductKind_Durable`	Durable product.
`e_ms_iap_ProductKind_Game`	A game.
`e_ms_iap_ProductKind_Pass`	A pass.
`e_ms_iap_ProductKind_UnmanagedConsumable`	A game managed consumable product, also known as an unmanaged consumable.

As explained above the product kinds can be combined using the **bit-wise or operator** following the example below:

```
var _consumableAndDurableType = e_ms_iap_ProductKind_Consumable |
 e_ms_iap_ProductKind_Durable;
```

The code above will make it so the `_consumableAndDurableType` variable will filter both consumables and durables.