

Coroutines with XDK Sample

# Description

This sample demonstrates the use of coroutines with the XDK. The code implements a few of the features from the coroutine technical specification including:

* Basic future type
* Basic awaitable type
* Overloading co\_await operator to use with existing types

It also shows how to use the awaitable types defined in the C++/WinRT base header for asynchronous programming.

# Using the sample

Coroutines are a compiler extension that requires the use of the /await compiler switch. This can be added in the “Additional options” section of the project properties when using Visual Studio. Coroutines requires at least Visual Studio 2015 Update 3 or Visual Studio 2017. This sample was built for, and works best with, Visual Studio 2017 (15.5 update) or later.

# Using the sample

The following actions are available in this sample.

|  |  |
| --- | --- |
| Action | Gamepad Control |
| Launch account picker | A | |
| Query users available on the console | X | |
| Launch virtual keyboard | Y | |
| Exiting the sample | Left Trigger + Right Trigger + Right Bumper | |

Each operation demonstrates a different mechanism for using awaitable types and coroutines to handle program flow.

## Handling the account picker with an overloaded co\_await operator

The unary **co\_await** operator is used to request the suspension of a coroutine. Execution is returned to the function that called the coroutine. This operator can only be used with specific types called **awaitable** types. An awaitable type must contain the following public member functions.

* **await\_ready** – This is called to determine whether suspension is necessary when the co\_await operator is used to suspend a coroutine. If an operation finishes more quickly than expected this function can return **true** to signal there’s no need to suspend. Otherwise, it can return **false** to suspend the coroutine.
* **await\_suspend** – This function is called when the coroutine is suspending. If **await\_ready** returns **true** this function will not be called. It can be used to handle any operations that are necessary, like storing information in the awaitable object, before the coroutine is suspended.
* **await\_resume** – When a coroutine is resumed this function determines the value returned by the co\_await operator.

The API to open the account picker returns an **IAsyncOperation<AccountPickerResult^>^** object, but this is not an awaitable type. A global **operator co\_await** overload is provided by client code to allow the co\_await operator to be used with this type of object. It provides a substitute object, an **accountpicker\_await\_adapter**, that is awaitable to handle suspension logic. The **await\_ready** function checks the status of the **IAsyncOperation**; if it is already completed, then there is no need to suspend the coroutine and it can run to completion. The **await\_suspend** function uses the **Completed** event of the IAsyncOperation to create a new context to continue the coroutine once the account picker has been closed. The implementation of **accountpicker\_await\_adapter** in this sample is based on the **winrt::accountpicker\_await\_adapter** struct defined in the C++/WinRT projection headers shipped with the XDK.

## Handle virtual keyboard with a custom future type

The signature of a coroutine looks identical to any other subroutine; it has a return type, a name, and an argument list. The only difference is that the return type must be a **future** type. The **std::future** type can be used and is sufficient for most purposes. This sample implements a basic future type, **awaitable\_future**, which can be used as a reference for other implementations. Unlike awaitable types, a future type does not require a specific public interface. Instead, a future type must provide a struct with the name **promise\_type** within its own scope. The promise\_type must provide the following public interface.

* **get\_return\_object** – This function is called when a coroutine is called, but before the body of the coroutine begins executing. It handles the construction of the future object that is returned to the caller if the coroutine is suspended.
* **(Optional) get\_return\_object\_on\_allocation\_failure** – Providing this function allows you to gracefully handle the case in which the there is no memory available for the coroutine context when it is called.
* **initial\_suspend** – This is called before the body of the coroutine begins executing to determine if it should immediately suspend and return execution to the caller. If this function returns **true** the coroutine will begin execution. If it returns **false**, the coroutine will be suspended before it begins execution, but after the future object has been created.
* **final\_suspend** – This is called when execution runs to the end of a coroutine or encounters a **co\_return** statement. If this function returns **true** the coroutine context will be immediately released along with the promise\_type invalidating any data it contains. If this function returns **false** the data held by the promise\_type will remain valid for the caller to query, but the caller will also be responsible for cleaning up the coroutine context.
* **return\_value** OR **return\_void** – The promise type must implement *one and only one* of these functions. If the coroutine uses a **co\_return** statement to return a value to the caller, then that value is passed to **return\_value** to be stored in the promise\_type for the caller to query. If there is no co\_return statement, or a co\_return statement without a value, then **return\_void** must be implemented. In either case, the function is called immediately before **final\_suspend** when a coroutine is finished executing.
* **(Optional) operator new** – If this is provided in the promise\_type, then it will be used to allocate memory for the coroutine context instead of **::operator new**. This is useful for tracking memory usage or providing an optimized allocator.
* **(Optional) operator delete** – If an overloaded operator new is provided, this must be provided to free memory after the corotuine context is released.

Unlike the **accountpicker\_await\_adapter**, the **keyboard\_await\_adapter** provided by client code does not automatically resume the coroutine once the asynchronous TCUI function is completed. Instead, it relies on the caller to resume the coroutine at the appropriate time.

## Querying Users collection with multiple threads in the same function scope

Querying User::Users may take a very long time and cause hitching if it is access on the rendering thread. The C++/WinRT projection headers contain a number of awaitable types that can be used to seamlessly handle threading within a single function even in a game that does not utilize the C++/WinRT projections. By using an instance of a **resume\_background** object with the co\_await operator the function that called the coroutine immediately resumes execution, and a new thread is created to also continue executing the coroutine immediately. This offloads accessing User::Users onto a background thread in the same scope of the rest of the body of the coroutine. Source code for other useful awaitable types is available in base.h in the C++/WinRT projection headers shipped with the XDK.

# Update history

February 2018 – Initial release

# Privacy Statement

When compiling and running a sample, the file name of the sample executable will be sent to Microsoft to help track sample usage. To opt-out of this data collection, you can remove the block of code in Main.cpp labeled “Sample Usage Telemetry”.

For more information about Microsoft’s privacy policies in general, see the [Microsoft Privacy Statement](https://privacy.microsoft.com/en-us/privacystatement/).