

Asynchronous Programming Sample

*This sample is compatible with the Microsoft Game Development Kit (June 2020)*

# Description

This sample shows how to use **XAsync**, **XTaskQueue**, and **XAsyncProvider** to implement asynchronous programming and task handling in various ways. While it is required to use XAsync with GDK asynchronous functions, the libraries are powerful on their own and fully portable between Windows 10, Xbox One, and Project Scarlett.

The sample UI provides the interface to run example tests and show output. See **Implementation Notes** below for in-depth descriptions of each test and how to follow the code for the cases.

# Building the sample

If using an Xbox One devkit, set the active solution platform to Gaming.Xbox.XboxOne.x64.

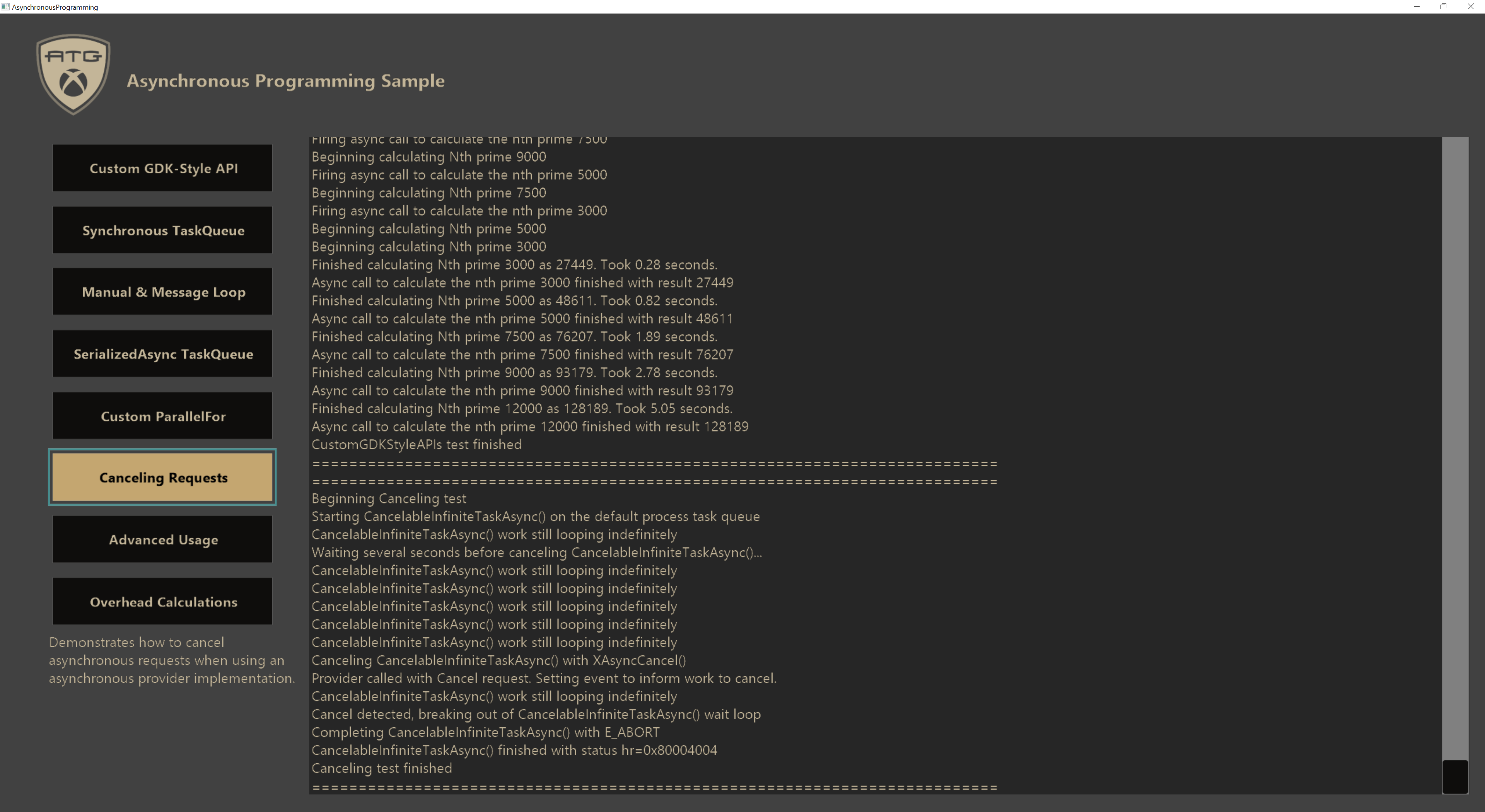
If using Project Scarlett, set the active solution platform to Gaming.Xbox.Scarlett.x64.

If using Windows 10, set the active solution platform to Gaming.Desktop.x64.

*For more information, see* Running samples*, in the GDK documentation.*

# Using the Sample

Simply use the buttons on the main screen to launch the different test scenarios for different usages of **XAsync**/**XTaskQueue**/**XAsyncProvider** libraries.



# Implementation Notes

The sample provides 8 different test scenarios to cover the different features of **XAsync**, **XTaskQueue**, and **XAsyncProvider**.

*For more information about these libraries, please see the GDK documentation. In short,* ***XAsync*** *provides methods for running and managing asynchronous tasks,* ***XTaskQueue*** *provides methods for managing the execution context/behavior of asynchronous tasks, and* ***XAsyncProvider*** *provides advanced management of the overall asynchronous task process plumbing.*

All usage and tests are implemented in the class **XAsyncExamples** except for the one case of integrating into the Windows Message Loop for one of the tests. Each test is in a function called “**StartTest\_[TestName]**”, so that is a good starting place to take a look at the implementations.

Task queues are created in **XAsyncExamples::CreateTaskQueues()** and threads are spawned in **XAsyncExamples::CreateThreads()**;. Shutdown logic for task queues is dependent on the port modes of the task queue, so see **XAsyncExamples::ShutdownTaskQueues()** for implementation details. When a task queue has a manual port, that port must continue to be dispatched until it returns since **XTaskQueueTerminate()** adds events to each port.

**RunTask()** is a common helper method that implements the standard **XAsyncBlock** creation and setup and then starts the async invocation with **XAsyncRun()**. It greatly simplifies executing tasks by only requiring the specification of the task queue, the work callback, and optionally the completion callback.

More information about each test is iterated below:

**Custom GDK Style API Test**

This test demonstrates how to setup asynchronous methods in the style of GDK API functions. The invocation and results process is identical to how an asynchronous GDK API method would function.

Task Queues:

* m\_taskQueue\_CustomGDKStyleAPIs
  + Work Port: ThreadPool
  + Completion Port: Manual

**NthPrimeAsync()** and **NthPrimeAsyncResult()** are the two methods implemented in the GDK Async style. Both functions require setting up an **XAsyncBlock** and passing that in the start the asynchronous task process. The asynchronous provider is implemented with **NthPrimeAsync()** for the plumbing and the result of the operation is returned from **NthPrimeAsyncResult()**.

The test function sets up 5 asynchronous calls to **NthPrimeAsync()** and reports the results in the completion callback. Since the task queue used has a ThreadPool work port, these 5 requests run in parallel on the system thread pool. For the Manual completion port, the callbacks are dispatched by a call to **XTaskQueueDispatch()** in **XAsyncExamples::Update()**. Callbacks on the completion port are executed in the order that they are enqueued.

**Synchronous TaskQueue Test**

This test demonstrates how asynchronous tasks can be invoked synchronously based solely on the specified task queue within the **XAsyncBlock** struct. In addition, this test also shows how the process default task queue can be changed.

Task Queues:

* m\_taskQueue\_SynchronousTaskQueue
  + Work Port: Immediate
  + Completion Port: Immediate
* Process Default Task Queue
  + Work Port: ThreadPool
  + Completion Port: ThreadPool

The test function first overrides the process default task queue to be a custom one with both ports set to Immediate. Immediate ports cause callbacks to be executed immediately when attempting to enqueue them instead of enqueuing them for later dispatching. The test also uses a helper function called **ParallelExecute()** which starts a callback a specified number of times on the specified task queue and then waits for all those tasks to complete.

The entire test is first moved to the system thread pool via **RunTask()** using a cached handle to the system default task queue. The purpose of this is to allow the test to run in the background without freezing the application. Then, 20 tasks are executed in parallel with the cached process default task queue handle which uses ThreadPool ports. Finally, the same 20 tasks are executed in the example same manner, but switching the task queue to the test one with ports set to Immediate. You will notice in the output that the asynchronous work completes quickly due to parallelization, but the synchronous version has to execute each task one at a time.

**Manual & Message Loop Test**

This test demonstrates how to use a manual task queue and how to integrate that task queue with the Windows Message Loop for completion callbacks. Several threads are used to dispatch the Manual work port in parallel and a task queue monitor callback is used to send a notification to the Windows Message Loop to process one Manual completion callback.

Task Queues:

* m\_taskQueue\_ManualAndMessageLoop
  + Work Port: Manual
  + Completion Port: Manual

There are several locations of interest for this test in code what aren’t just within **StartTest\_ManualAndMessageLoop()**. Of note, see **WndProc()**, **Notify\_TaskQueueMonitor()**, **ThreadProc\_ManualAndMessageLoop()**, **Monitor\_ManualAndMessageLoop()**, **CreateTaskQueues()**, and **ShutdownTaskQueues()**.

When the task queue for this test is created in **CreateTaskQueues()**, notice that a monitor is created for the task queue as well. This monitor is executed whenever a new callback is added to a port on the task queue. Whenever that monitor is executed, it will use **PostMessage()** with *WM\_TASKQUEUEMONITOR* whenever a callback for the completion port is detected. **WndProc()** is setup to check for the *WM\_TASKQUEUEMONITOR* event and calls into **Notify\_TaskQueueMonitor()** which dispatches one callback from the completion port within the context of the Windows Message Loop.

Finally, the test function is a simple driver which starts 20 asynchronous requests upon the test task queue with both work and completion callbacks specified.

**SerializedAsync TaskQueue Test**

This test demonstrates how to setup and use a task queue with SerializedThreadPoolports and how those ports function.

Task Queues:

* m\_taskQueue\_SerializedAsync
  + Work Port: SerializedThreadPool
  + Completion Port: SerializedThreadPool

A port set to SerializedThreadPool automatically dispatched callbacks on the system thread pool, but only one at a time. Callbacks are dispatched in the order they are enqueued. This behavior is the same as if you would use a Manual port and dispatch one callback at a time on only one thread. However, the system thread pool doesn’t require the same thread to be used each time.

Because a SerializedThreadPool port only runs one callback at a time, you can have dependencies between the different callbacks of the asynchronous work. The test function exploits this by enqueuing 5 different tasks upon the task queue where each task uses the result from the previous task and changes the data for the next task.

**Custom ParallelFor Test**

This test demonstrates one way that a ParallelFor function can be implemented. A manual task queue is used to perform the work and a Prefix Sum calculation is performed to test the performance.

Task Queues:

* m\_taskQueue\_ParallelFor
  + Work Port: Manual
  + Completion Port: Immediate

To try to get max parallel performance, a thread is spawned for each logical processor and each thread is initialized to have an affinity for one processor. **ThreadProc\_SingleCore()** can call **XTaskQueueDispatch()** in at the same time to get parallelization and each thread will get a different task.

The prefix sum is calculated in two ways to test synchronous and asynchronous behavior. To give the best possible comparison, a tight loop is implemented for the synchronous case and a different parallel-optimized algorithm is used for the asynchronous case. The timings for the cases are reported in the log.

**Canceling Requests Test**

This test demonstrates how to cancel asynchronous tasks. Not all asynchronous tasks can be canceled as the canceling behavior must be manually handled within the provider for the asynchronous task. As a result, simple tasks run via **XAsyncRun()** (used by **RunTask()**) cannot be canceled. To demonstrate a cancelable task, a custom async provider function is used (much like the GDK-Style test) with a cancel path implemented.

Task Queues:

* Default Process Task Queue
  + Work Port: ThreadPool
  + Completion Port: ThreadPool

See **CancelableInfiniteTaskAsync()** for the implementation of an asynchronous method with a custom provider. This async method doesn’t return data, but runs infinitely unless it is canceled. The **XAsyncOp::Cancel** case is implemented to set an event which the infinite runtime can detect. Once that event is set, the provider knows to complete with **XAsyncComplete()** using *E\_ABORT*.

The test case function starts a call to **CancelableInfiniteTaskAsync()** and also starts another task to call **XAsyncCancel()** after some time. Watch the output log to see the test in action.

**Advanced Usage Test**

This test demonstrates some uncommon advanced features, including composite task queues, duplicated task queue handles, and using waiters and delayed dispatching of events. In this test, all the task queues and other test data are created and destroyed within the test function to show the whole process.

The test runs through a test of composite queues first. A composite queue is a task queue whose ports are composed of ports from other task queues. A normal task queue is first created with the ThreadPool as the work port and Manual set as the completion port. Then, a composite task queue is created that utilizes the work port of the first task queue as both of its own ports. This makes it so that both work and completion callbacks run on the same port of the original queue.

Next, the test shows how to duplicate queue handles. Whenever a task queue handle is duplicated, it must be closed that many times to finalize cleanup. Duplicate handles function identically to the original handle and just increase ownership over the resource.

Finally, waiters and delayed dispatching is tested. A waiter is a way to submit a callback for execution multiple times, but the submission only happens each time a specific event is triggered. This is useful for automatic execution of events. Delayed dispatching is the submission of a callback directly to one of the ports of a task queue with an optional delay. Normal async submission requires an **XAsyncBlock**, but **XTaskQueueSubmitCallback**/**XTaskQueueSubmitDelayedCallback** submit callbacks directly to the task queue instead. They’re used internally by async providers normally.

**Overhead Calculations Test**

This test calculates timings of various overheads in utilizing **XAsync**/**XTaskQueue**/**XAsyncProvider**. It can be useful to understand the different overheads of utilizing asynchronous methods.

The overheads are calculated for three different scenarios with multiple cases for each scenario. See the chart below for descriptions of each timing.

|  |  |
| --- | --- |
| Timing Name | Description |
| XAsyncRun\_InvokeToWork (Process Default Task Queue) | Times how long it takes between invoking **XAsyncRun()** and when the work callback starts on the process default task queue. |
| XAsyncRun\_InvokeToWork (Manual Task Queue) | Times how long it takes between invoking **XAsyncRun()** and when the work callback starts using a manual queue with already-waiting threads. |
| XAsyncRun\_WorkToCompletion (Process Default Task Queue) | Times how long it takes between leaving the work callback schedules from **XAsyncRun()** and when the completion callback starts using the process default task queue. |
| XAsyncRun\_WorkToCompletion (Manual Task Queue) | Times how long it takes between leaving the work callback schedules from **XAsyncRun()** and when the completion callback starts using a manual queue with already-waiting threads. |
| ParallelFor\_InvokeToWork | Times how long it takes for the custom **ParallelFor()** method implemented in this sample to get from invocation time to the start of one of the work callbacks. |
| ParallelFor\_InvokeToReturn | Times how long it takes for the custom **ParallelFor()** method implemented in this sample to complete the whole invocation of the method with no substantial work. |
| GDKAsyncStyle\_TimeInProviderAverage (Process Default Task Queue) | Times the average time spent in a custom provider method when using the process default task queue. |
| GDKAsyncStyle\_TimeInProviderAverage (Manual Task Queue) | Times the average time spent in a custom provider method when using a manual queue with already-waiting threads. |
| GDKAsyncStyle\_TimeInProviderOverall (Process Default Task Queue) | Times the overall time spent in a custom provider method when using the process default task queue. |
| GDKAsyncStyle\_TimeInProviderOverall (Manual Task Queue) | Times the overall time spent in a custom provider method when using a manual queue with already-waiting threads. |
| GDKAsyncStyle\_InvokeToWork (Process Default Task Queue) | Times how long it takes to go from invocation to the start of the work callback when using a custom provider with the system default task queue. |
| GDKAsyncStyle\_InvokeToWork (Manual Task Queue) | Times how long it takes to go from invocation to the start of the work callback when using a custom provider with a manual queue with already-waiting threads. |
| GDKAsyncStyle\_WorkToCompletion (Process Default Task Queue) | Times how long it takes between the work and completion callbacks of a custom provider method when using the process default task queue. |
| GDKAsyncStyle\_WorkToCompletion (Manual Task Queue) | Times how long it takes between the work and completion callbacks of a custom provider method when using a manual task queue with already-waiting threads. |

The timings are re-calculated each run and will be output in the log.

# A Note on SMT and Time-Sensitive Threads

**SMT (Simultaneous Multi-Threading)**

Project Scarlett devices support SMT to allow the logical core count to double. However, the increased power draw causes the frequency of the cores to drop with SMT enabled. This means that the title can choose to have 7 faster cores, or 14 slightly slower cores with some sharing considerations.

More information can be found in documentation in the section titled *“Xbox One vs. Project Scarlett CPU & Memory”*.

To enable SMT in this sample, open the MicrosoftGame.config file and uncomment the <VirtualMachine> section in it.

To highlight the difference in CPU behavior, here are some timings when testing a couple different tests with SMT on and off:

|  |  |  |
| --- | --- | --- |
|  | ParallelFor PrefixSum Async Time | 5000th Prime Calculation Sync/Parallel |
| SMT on | Ranges from 100ms to 125ms | Consistent 0.47s sync, 3.88~9.20s parallel, ~9.3s total |
| SMT off | Ranges from 140ms to 175ms | Consistent 0.44s sync, 0.44~1.14s parallel, ~4.5s total |

Observances:

* With SMT on, algorithms designed to be parallelized well perform better, but just filling every core with intense calculations can harm performance.
* With SMT off, parallel algorithms don’t get as much of a performance benefit with less cores, but single-threaded type work is faster and the cores are more resistant to performance loss with full utilization of intense work.

It’s ultimately up to the developer to determine whether SMT is right for the title.

**Time-Sensitive Threads**

Not every GDK API function is safe to call on threads that should run quickly without blocking. These threads are called “time-sensitive”. To support ensuring that GDK API functions won’t block these threads, the time-sensitive APIs were added to allow for marking the time-sensitive threads. After being marked, calling a potentially slow GDK API function will trigger an assert for the developer to know about. These asserts can be progressed past by resuming execution.

More information can be found in documentation in the section titled *“Time-sensitive threads”*.

To test in this sample, uncomment the call to **XThreadSetTimeSensitive()** in **Sample::Initialize()**. This will enable the time-sensitive checks for this thread. After enabling, the asserts will fire in two different tests, explained below:

* Synchronous Task Queue Test
  + **XTaskQueueSetCurrentProcessTaskQueue()**
    - This function is used to demonstrate how to change the process’s default task queue. Normally this should be done at a loading time to avoid a potential slowdown, but it’s done in this test to show functionality.
* Advanced Usage Test
  + **XTaskQueueCreate()**
    - In the sample, most task queues are created early in initialization before marking the thread as time sensitive. For this test, the task queues are created directly in the test. This function is not time-sensitive safe as it can take a short time to create.
  + **XTaskQueueGetPort()**
    - Getting the ports of a task queue is not a fast operation and is considered to really only be done as part of initialization like when calling **XTaskQueueCreate()**.
  + **XTaskQueueCreateComposite()**
    - This is the same as calling **XTaskQueueCreate()** with all the same considerations. The only different is which ports the new task queue is using.
  + **XTaskQueueRegisterWaiter()**
    - Registering a waiter is another slow operation that should be reserved for loading times or brought off a time-sensitive thread to avoid slowdowns or hitching.

The functions above that trigger the time-sensitive assert should generally be reserved for calling at a load time as task queues generally stick around a long time. When doing this, the thread can temporarily have its time-sensitive status removed with another call to **XThreadSetTimeSensitive(false)** during loading sequences. If needing to do this during gameplay or other active times, the calls can be moved to a non-time-sensitive thread to avoid any noticeable affect on application performance.

# Update history

**Initial Release:** Microsoft Game Development Kit (May 2020)

June 2020: Fixed lambda capture bug causing tests to not finish. Rendering size and lost device known issues fixed and section removed. Added SMT and Time-Sensitive Threads information to readme and Time-Sensitive Thread calls to sample.

# 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/).