# On ways of finding out when a C++/WinRT IAsyncAction has run to completion

devblogs.microsoft.com/oldnewthing/20220511-00

May 11, 2022

Raymond Chen

A customer had a C++/WinRT coroutine that just kept on running until you told it to stop.

```
struct Widget : winrt::implements<Widget, IWidget>
{
private:
    winrt::Windows::Foundation::IAsyncAction m_work;

    IAsyncAction DoStuffUntilCancelled()
    {
        co_await winrt::resume_background();

        auto cancellation = co_await winrt::get_cancellation_token();
        cancellation.enable_propagation();

        // Run this loop until cancelled.
        while (!cancellation()) {
            one();
            co_await two();
            co_await three();

            // pause a little bit before going again
            co_await 1s;
        }
    }

public:
    void Start()
    {
        m_work = DoStuffUntilCancelled();
    }

    void Stop()
    {
        m_work.Cancel();
        ... wait until DoStuffUntilCancelled
            has definitely stopped ...
    }
};
```

The idea is that you can `Start()` the Widget to make it begin doing something in the background, and you can `Stop()` it to make that thing stop.

Let's assume rules that say you can `Start()` a Widget at most once, and you cannot `Stop()` without a preceding `Stop()`. Because that's not really the point of the exercise.

The point of the exercise is to figure out how to implement the "wait until `DoStuffUntil-Cancelled` has definitely stopped" part.

In the C++/WinRT implementation of `IAsyncAction`, when the coroutine is cancelled, polling for cancellation will begin returning `true`, and any `co_await` operation will raise a cancellation exception. Furthermore, since we enabled cancellation propagation, if the coroutine is cancelled while it is suspended in a `co_await`, the cancellation will be propagated to the thing being `co_await`ed.

The choice of expressing the background activity in the form of an `IAsyncAction` limits your ways of communicating with the coroutine. The only thing you can do to alter the behavior of a running `IAsyncAction` is to `Cancel()` it, which triggers the cancellation flow summarized above.

So we'll just have to deal with the cancellation.[1]

One thing we could do is give the coroutine an event handle to signal, and have the coroutine set the event either when it chooses to exit on its own, or when it observes the cancellation from within the coroutine:

```
IAsyncAction DoStuffUntilCancelled(HANDLE event)
{
  try {
    co_await winrt::resume_background();

    auto cancellation = co_await winrt::get_cancellation_token();
    cancellation.enable_propagation();

    // Run this loop until cancelled.
    while (!cancellation()) {
        one();
        co_await two();
        co_await three();

        // pause a little bit before going again
        co_await 1s;
    }
    SetEvent(event);
  } catch (hresult_canceled const&) {
    SetEvent(event);
    throw;
  }
}
```

Of course, this is more easily expressed as an RAII type.

```
IAsyncAction DoStuffUntilCancelled(HANDLE event)
{
    auto setOnExit = wil::SetEvent_scope_exit(event);

    co_await winrt::resume_background();

    auto cancellation = co_await winrt::get_cancellation_token();
    cancellation.enable_propagation();

    // Run this loop until cancelled.
    while (!cancellation()) {
        one();
        co_await two();
        co_await three();

        // pause a little bit before going again
        co_await 1s;
    }
}
```

This alternative version has the advantage of also working if the coroutine is destroyed before running to completion. (C++/WinRT coroutines don't behave like this, but coroutines from some other library might.)

Another pattern you often run into is wanting to stop the background coroutine when the Widget destructs. For that, you can take advantage of the `final_release` extension point, which allows you to run extra code prior to destruction, possibly even itself a coroutine.

```
struct Widget : winrt::implements<Widget, IWidget>
{
private:
    awaitable_manual_reset_event m_done;
    winrt::Windows::Foundation::IAsyncAction m_work;

    IAsyncAction DoStuffUntilCancelled(awaitable_manual_reset_event done)
    {
        auto setOnExit = wil::scope_exit([&] { done.set(); });

        co_await winrt::resume_background();

        auto cancellation = co_await winrt::get_cancellation_token();
        cancellation.enable_propagation();

        // Run this loop until cancelled.
        while (!cancellation()) {
            one();
            co_await two();
            co_await three();

            // pause a little bit before going again
            co_await 1s;
        }
    }

public:
    void Start()
    {
        m_work = DoStuffUntilCancelled();
    }

    static winrt::fire_and_forget final_release(std::unique_ptr<Widget> widget)
    {
        if (widget->m_work) {
            widget->m_work.Cancel();

            // make sure coroutine has exited before we destruct
            co_await widget->m_done;
        }
    }
};
```

[1] Another way out is to change the game. Instead of returning an `IAsyncAction`, return an Operation object, with methods to request a stop, and to wait for the stop to complete.

Raymond Chen

**Follow**