# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>2</td>
</tr>
<tr>
<td>Background Information</td>
<td>2</td>
</tr>
<tr>
<td>Basic Example</td>
<td>3</td>
</tr>
<tr>
<td>Specifications and Usage</td>
<td>4</td>
</tr>
<tr>
<td>BeginInvoke()</td>
<td>4</td>
</tr>
<tr>
<td>EndInvoke()</td>
<td>5</td>
</tr>
<tr>
<td>IAsyncResult</td>
<td>5</td>
</tr>
<tr>
<td>Advanced Examples</td>
<td>6</td>
</tr>
<tr>
<td>Polling IAsyncResult.IsCompleted</td>
<td>6</td>
</tr>
<tr>
<td>Using a Callback Delegate</td>
<td>7</td>
</tr>
<tr>
<td>Conclusion</td>
<td>10</td>
</tr>
<tr>
<td>Appendix</td>
<td>11</td>
</tr>
<tr>
<td>Appendix 1: Reference Materials</td>
<td>11</td>
</tr>
<tr>
<td>Appendix 2: Full Code Examples</td>
<td>12</td>
</tr>
<tr>
<td>Basic Example</td>
<td>12</td>
</tr>
<tr>
<td>As Little Blocking As Possible Example</td>
<td>13</td>
</tr>
<tr>
<td>Callback Example</td>
<td>14</td>
</tr>
</tbody>
</table>
C# Asynchronous Programming Model

INTRODUCTION

The goal of this guide is to introduce and to demonstrate the asynchronous capabilities of C#. There are several ways to asynchronously perform operations in C#. This guide will focus on the use of C# delegates to spawn and interact with threads. This guide will assume that the reader has some knowledge of the basics of both C# programming and concurrent programming (C# or otherwise). References for both topics are listed in Appendix 1.1.

BACKGROUND INFORMATION

As mentioned above, this guide will demonstrate the use of C# delegates to asynchronously perform operations. A C# delegate is defined as follows by the Microsoft Developer Network C# Programming Guide:

A delegate is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance.

Basically, a C# delegate is a wrapper containing a number of method references that can be invoked by the delegate. The methods contained in the delegate must have the proper signature, i.e. the same return type and parameter listing. Below is a simple example illustrating the declaration, creation, and invocation of a delegate.

```csharp
namespace Examples
{
    class DelegateExample
    {
        public delegate string BasicDelegate(int arg1, int arg2);

        public string DelegateMethod(int arg1, int arg2)
        {
            return "" + arg1 + " -- " + arg2;
        }

        static void Main(string[] args)
        {
            DelegateExample ex = new DelegateExample();
            BasicDelegate dg = new BasicDelegate(ex.DelegateMethod);
            string val = dg.Invoke(32, 18);
            Console.WriteLine(val);
            Console.WriteLine("Press any key to continue...");
            Console.ReadKey();
        }
    }
}
```
C# Asynchronous Programming Model

Take note of the delegate declaration, `BasicDelegate`, and its construction below in the Main method where it takes the method, `DelegateMethod`. This construction can take in any method with the same method signature, (String `methodName`(int, int)).

**BASIC EXAMPLE**

To asynchronously perform an operation in C#, the operation will need to be wrapped in a method. Then that method will be wrapped in an appropriate delegate (just like in the example above). However, instead of calling Invoke(), which invokes the method call synchronously, we need to call BeginInvoke() to invoke the method asynchronously. BeginInvoke() will return an object implementing the IAsyncResult interface. The returned IAsyncResult is how we interact with the asynchronous method call. Using the returned IAsyncResult, we can access many properties associated with the asynchronous call including whether or not the asynchronous operation is completed. Also, more importantly, the EndInvoke() delegate method takes in an IAsyncResult. EndInvoke() will block until the asynchronous method call associated with the passed-in IAsyncResult is completed and retrieve any values or thrown errors returned by it.

```csharp
9  class DelegateExample
10  {
11     public delegate String BasicDelegate(int arg1, int arg2);
12  
13     public String DelegateMethod(int arg1, int arg2)
14     {
15         return "" + arg1 + " -- " + arg2;
16     }
17  
18     static void Main(string[] args)
19     {
20         DelegateExample ex = new DelegateExample();
21         // Create instance of delegate. This delegate can now be invoked.
22         // This will call the method inside the delegate.
23         BasicDelegate dg = new BasicDelegate(ex.DelegateMethod);
24         // Call BeginInvoke on the created delegate. Spawn
25         // a new thread to perform the call to DelegateMethod.
26         // Returns an IAsyncResult.
27         IAsyncResult result =
28             dg.BeginInvoke(32, 18, null, null);
29         // Call EndInvoke on the created delegate. Takes in
30         // an IAsyncResult as its only argument. Returns the value
31         // returned by the encapsulated method. In this case: "32 -- 18"
32         String val =
33             dg.EndInvoke(result);
34         Console.WriteLine(val);
35         Console.WriteLine("Press any key to continue...");
36         Console.ReadKey();
37     }
38  }
```
Above, Figure 2.1, is a very simple example using an asynchronous delegate.

The most important thing to take from the example is how the IAsyncResult, result, is used to bridge the calls to BeginInvoke() and EndInvoke(). A constructed delegate can spawn as many threads as the operating system will allow by calling BeginInvoke() repeatedly, however, the delegate will not maintain any information about those processes. The returned IAsyncResult is the only means we have to interact with an asynchronous process.

SPECIFICATIONS AND USAGE

This section will cover the specifications of the methods and interfaces used in C#’s delegate-based asynchronous programming. It will also explain some of the nuances that are associated with the different pieces.

BEGININVOKED()

**Signature:** BeginInvoke([delegate args…], AsyncCallback cb, Object asyncState): IAsyncResult

- [delegate args…] – Arguments to be passed to the async method call. This is specified in the calling delegate. Recall in Figure 2.1 when our delegate called BeginInvoke(), the first two arguments were ints. These two ints matched the signature of our BasicDelegate.

- AsyncCallback cb – A delegate that will be invoked once the asynchronous operation completes. AsyncCallback is a pre-defined delegate that takes in a method with the following signature: [callbackMethod](IAsyncResult r): void, where callbackMethod is the name of the method. This will be explained a bit more later. **Can be null.**

- Object asyncState – A user-defined object used to contain some state that can be accessed within the IAsyncResult. Anything that can access the IAsyncResult created by BeginInvoke has access to this object and can view its contents. This is very handy when using the AsyncCallback. **Can be null.**

- Returns IAsyncResult – This is used to interact with the asynchronous invocation started by BeginInvoke. It is useful to note that an instance of the concrete type AsyncResult is returned as an IAsyncResult. AsyncResult has some more useful properties along with what’s already available in IAsyncResult (see the IAsyncResult section for more information).

The last two parameters are optional and can both be null.
**EndInvoke()**

**Signature:** EndInvoke(IAsyncResult r): [delegate return type]

- IAsyncResult r – The IAsyncResult associated with the asynchronous operation you wish to end.
- [delegate return type] – EndInvoke will return the return type specified by the calling delegate. Recall, in Figure 2.1, that the call to EndInvoke returned a String. This matches up with its caller’s, an instance of BasicDelegate, return type.

It is very important to note that EndInvoke will block until the asyn operation is completed. There are ways to check to see if the IAsyncResult’s operation is finished prior to calling EndInvoke to ensure that no blocking occurs (see the IAsyncResult section for more information).

EndInvoke should be called exactly once for every IAsyncResult, even if there’s no need to get the results of the async call. The .NET framework, upon which C# runs, will create a new thread when BeginInvoke is called. This thread will take up system resources until the thread is destroyed or .NET garbage collects it. EndInvoke destroys the thread, or at least marks it as complete to help in garbage collection. If this doesn’t happen, .NET will wait for ten minutes of inactivity from the thread to reclaim that memory. On the other side of it, the behavior for calling EndInvoke more than once is undefined and C# will throw an InvalidOperationException when this happens.

**IAsyncResult**

**IAsyncResult Interface**

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AsyncState</td>
<td>Object</td>
<td>Gets a user-defined object that qualifies or contains information about an asynchronous operation.</td>
</tr>
<tr>
<td>AsyncWaitHandle</td>
<td>WaitHandle</td>
<td>Gets a WaitHandle that is used to wait for an asynchronous operation to complete.</td>
</tr>
<tr>
<td>CompletedSynchronously</td>
<td>Bool</td>
<td>Gets a value that indicates whether the asynchronous operation completed synchronously.</td>
</tr>
<tr>
<td>IsCompleted</td>
<td>Bool</td>
<td>Gets a value that indicates whether the asynchronous operation has completed.</td>
</tr>
</tbody>
</table>
This interface defines what ways in which we can interact with an asynchronous method call, including means for polling for operation completion (IAsyncResult.IsCompleted) and passing state between interactions (IAsyncResult.AsyncState).

ADVANCED EXAMPLES

Now that we’ve covered the basics and explained how all of the pieces work, this guide will demonstrate how to use them effectively through a few examples.

POLLING IASYNCRESULT.ISCOMPLETED

This example will demonstrate how we can use some of the properties in IAsyncResult so that we don’t have to block while waiting for the asynchronous operation to complete.

First, let’s set up the operation that we’ll want to asynchronously and create a delegate for it.

```csharp
namespace Examples
{
    class AsLittleBlockingAsPossibleExample
    {
        public delegate void LongOpDelegate();

        public void ReallyLongOperation()
        {
            System.Threading.Thread.Sleep(5000); //sleep for five seconds
        }
    }
}
```

Then let’s create a Main method to use them in. The example will run 25 “short operations” that will each last for 250 ms along with our “really long operation” which will last 5 seconds. The short operations will “execute” while our “really long operation” executes asynchronously. Once the long operation is completed, it is handled and then any other short operations are completed. If all of the short operations are completed before the long operation terminates, then the main thread will block on EndInvoke until it does, hence “AsLittleBlockingAsPossibleExample” for the class name.
C# Asynchronous Programming Model

```csharp
static void Main(string[] args)
{
    // prepare delegate to do ReallyLongOperation
    AsLittleBlockingAsPossibleExample example = new AsLittleBlockingAsPossibleExmple();
    LongOpDelegate myDelegate = new LongOpDelegate(example.ReallyLongOperation);

    Console.WriteLine("Starting really long operation, hold onto your butts!");

    // asynchronously invoke ReallyLongOperation
    IAsyncResult result = myDelegate.BeginInvoke(null, null);

    // act busy until ReallyLongOperation is done
    int numShortOperations = 25;
    int i = 0;

    // work until either the async op is done and we can do work on
    // the results OR we can't do any more work.
    while (!result.IsCompleted && i < numShortOperations)
    {
        Console.WriteLine("Operation " + i + " done!");
        System.Threading.Thread.Sleep(250);
    }

    // call EndInvoke and clean up resources. May block if we ran out of other work
    myDelegate.EndInvoke(result);
    Console.WriteLine("Really long operation is done!");

    // finish any work that didn't get done.
    for (; i < numShortOperations; i++)
    {
        Console.WriteLine("Operation " + i + " done!");
        System.Threading.Thread.Sleep(250);
    }

    Console.WriteLine("Press any key to continue...");
    Console.ReadKey();
}
```

**USING A CALLBACK DELEGATE**

This example will illustrate a bit more of the functionality built into asynchronous delegates. Specifically, we’ll look at how a callback can be used to handle any code that needs to be completed after the asynchronous operation has finished and at how we can pass state using the AsyncState property of IAsyncResult.

We’ll use the same ReallyLongOperation and LongOpDelegate we used in the above example.

Then let’s create our Callback method. This is the code that will be run once our async operation is completed.
C# Asynchronous Programming Model

```csharp
public void Callback(IAsyncResult cb)
{
    Console.WriteLine("I'm the callback! The ReallyLongOperation is done!");
    Console.WriteLine("AsyncState = " + cb.AsyncState.ToString());
}
```

Take note of the parameter list for Callback and return type. It takes in a single IAsyncResult that will be passed in automatically when the callback is invoked and returns nothing. The Callback method also prints out the contents of the AsyncState object contained in the passed-in IAsyncResult. More on that later.

Now, let’s put in a Main method and create all of our pieces.

```csharp
static void Main(string[] args)
{
    // prepare delegate to do ReallyLongOperation
    CallbackExample example = new CallbackExample();
    LongOpDelegate myDelegate = new LongOpDelegate(example.ReallyLongOperation);

    // prepare callback delegate
    // this delegate will invoke example.Callback (above)
    // when the async operation started by BeginInvoke (below)
    // completes.
    AsyncCallback cb = new AsyncCallback(example.Callback);

    // asyncState to be passed to the created IAsyncResult
    String asyncState = "I'm the asyncState!";
}
```

We’ve got our class, our delegate created with the ReallyLongOperation, our AsyncCallback created with our Callback method, and our asyncState object. Finally, let’s put them all together.
Line 46 is where all the magic happens. The instantiated ‘myDelegate’ is used to asynchronously invoke ReallyLongOperation with our callback and asyncState. The resulting output looks something like this.
The main thread is simulating other operations while awaiting the asynchronous operation to complete and the callback to be invoked. The callback is then invoked and the main thread finishes. Note that the asyncState string shows up in the callback output.

CONCLUSION

There you have it. These are the fundamentals for using asynchronous delegates in C#. Please note that this guide did not cover some of the more advanced usage such as synchronization and thread safety as they were out of the scope of this document. However, these things should definitely be considered when utilizing any asynchronous programming.
APPENDIX

APPENDIX 1: REFERENCE MATERIALS


Concurrent Programming Reference:
class DelegateExample
{
    public delegate string BasicDelegate(int arg1, int arg2);

    public string DelegateMethod(int arg1, int arg2)
    {
        return "" + arg1 + "--" + arg2;
    }

    static void Main(string[] args)
    {
        DelegateExample ex = new DelegateExample();

        // Create instance of delegate. This delegate can now be invoked.
        // This will call the method inside the delegate.
        BasicDelegate dg = new BasicDelegate(ex.DelegateMethod);

        // Call BeginInvoke on the created delegate. Spawn
        // a new thread to perform the call to DelegateMethod.
        // Returns an IAsyncResult.
        IAsyncResult result =
            dg.BeginInvoke(32, 18, null, null);

        // Call EndInvoke on the created delegate. Takes in
        // an IAsyncResult as its only argument. Returns the value
        // returned by the encapsulated method. In this case: "32--18"
        String val =
            dg.EndInvoke(result);

        Console.WriteLine(val);
        Console.WriteLine("Press any key to continue..."esture);
        Console.ReadKey();
    }
}
using System.Threading.Tasks;

namespace Examples
{
    class AsLittleBlockingAsPossibleExample
    {
        public delegate void LongOpDelegate();

        public void ReallyLongOperation()
        {
            System.Threading.Thread.Sleep(5000); // sleep for five seconds
        }

        static void Main(string[] args)
        {
            // prepare delegate to do ReallyLongOperation
            AsLittleBlockingAsPossibleExample example = new AsLittleBlockingAsPossibleExample();
            LongOpDelegate myDelegate = new LongOpDelegate(example.ReallyLongOperation);

            Console.WriteLine("Starting really long operation, hold onto your butts!");

            // asynchronously invoke ReallyLongOperation
            TaskResult result = myDelegate.BeginInvoke(null, null);

            // act busy until ReallyLongOperation is done
            int numShortOperations = 25;
            int i = 0;

            // work until either the async op is done and we can do work on the results OR we can't do any more work.
            while (!result.IsCompleted && i < numShortOperations)
            {
                Console.WriteLine("Operation " + i + " done!");
                System.Threading.Thread.Sleep(250);
            }

            // call EndInvoke and clean up resources. May block if we ran out of other work
            myDelegate.EndInvoke(result);
            Console.WriteLine("Really long operation is done!");

            // finish any work that didn't get done.
            for (; i < numShortOperations; i++)
            {
                Console.WriteLine("Operation " + i + " done!");
                System.Threading.Thread.Sleep(250);
            }

            Console.WriteLine("Press any key to continue...");
            Console.ReadKey();
        }
    }
}
namespace Examples
{
    class CallbackExample
    {
        public delegate void LongOpDelegate();

        public void ReallyLongOperation()
        {
            System.Threading.Thread.Sleep(5000); // sleep for five seconds
        }

        // The callback method takes an IAsyncResult, this is the same
        // reference that is returned when BeginInvoke is called (they refer
        // to the same async operation).
        public void Callback(IAsyncResult cb)
        {
            Console.WriteLine("I'm the callback! The ReallyLongOperation is done!");
            Console.WriteLine("AsyncState = " + cb.AsyncState.ToString());
        }

        static void Main(string[] args)
        {
            // prepare delegate to do ReallyLongOperation
            CallbackExample example = new CallbackExample();
            LongOpDelegate myDelegate = new LongOpDelegate(example.ReallyLongOperation);

            // prepare callback delegate
            // this delegate will invoke example.Callback (above)
            // when the async operation started by BeginInvoke (below)
            // completes.
            AsyncCallback cb = new AsyncCallback(example.Callback);

            // asyncState to be passed to the created IAsyncResult
            String asyncState = "I'm the asyncState!";

            Console.WriteLine("Starting really long operation, hold onto your butts!");

            // asynchronously invoke ReallyLongOperation
            IAsyncResult result = myDelegate.BeginInvoke(cb, asyncState);

            // act busy until ReallyLongOperation is done
            int i = 0;
            while (!result.IsCompleted)
            {
                Console.WriteLine("Operation " + i++ + " done!");
                System.Threading.Thread.Sleep(250);
            }

            // call EndInvoke and clean up resources
            myDelegate.EndInvoke(result);

            Console.WriteLine("Press any key to continue...".ToString());
            Console.ReadKey();
        }
    }
}