# ECEC 356 - Embedded Systems

# Lab 2 - prelab, lab, postlab

# 10/3/2014– due 10/17/2014

# PRE-LAB 2

read the following documents from the Microsoft robotics SDK (.\Microsoft Robotics Dev Studio 4\documentation):

Getting\_Started\_With\_RDS.pdf

RDS\_Reference\_Platform.pdf

Simulated\_Reference\_Platform.pdf

# LAB 2

## objectives:

* introduction to the robotics SDK, the simulation platform and reference robot
* introduction to the concurrency and coordination runtime

0. Open the reference platform application under the visual simulation environment folder in the robotics SDK: ".\Microsoft Robotics Dev Studio 4\bin\DssHost32.exe" /p:50000 /t:50001 /m:"samples\config\ReferencePlatform2011.simulation.manifest.xml"

verify that you are able to run the reference platform simulation tools. You may need to execute “build all samples” from the start menu. Navigate the robot through the provided scene, observing the different sensors. DEMONSTRATE

You can also run this from the start menu directly - be sure to try it both ways.

###### The remainder of lab 2 will introduce the concurrency and coordination runtime. The information and experimental steps for you to follow below are adapted directly from the robotics SDK help file. Implement the code fragments that are provided in each of the example steps below. One challenge is simply getting the code to run. Execute **every** line of code in the debugger. Another challenge is managing the code fragments from the 24 examples while keeping your solution readable and understandable to others. Be organized and neat!

Concurrency and Coordination Runtime (CCR) is a managed code library, a Dynamically Linked Library (DLL), accessible from any language targeting the .NET Common Language Runtime (CLR).

The CCR addresses the need of service-oriented applications to manage asynchronous operations, deal with concurrency, exploit parallel hardware and deal with partial failure. It enables the user to design applications so that the software modules or components can be loosely coupled; meaning they can be developed independently and make minimal assumptions about their runtime environment and other components. This approach changes how the user can think of programs from the start of the design process and deals with concurrency, failure and isolation in a consistent way.

### Problem Areas

* **Asynchrony** - When communicating between loosely coupled software components, like programs running across the network, or User Interface (UI) code communicating with the user input and the **file I/O subsystem**, asynchronous operations enable the code to scale better, be more responsive, and deal with failure across multiple operations. Asynchronous programming however, considerably reduces the readability of user code, since logic is often split between callbacks and the code that originates the operation. In addition, it is an almost impossible task to correctly handle failure across multiple outstanding operations.
* **Concurrency** - Code that needs to better utilize multiple execution resources, must be split into independent logical segments, that can run in parallel, and communicate when necessary to produce results from the combined execution. Often, that logical segment is captured by the thread OS primitive, that is nothing more than a long lived iteration. Because of thread performance implications on thread startup, the thread stays active for long periods of time. This forces a particular pattern. Code is structured as long sequences that use blocking or synchronous calls, and only deals with one thing at a time. Further, threads assume that the primary communication between them is shared memory, forcing the programmer to use very explicit, error-prone methods to synchronize access to that shared memory.
* **Coordination and Failure Handling** - Coordinating between components is where most of the complexity in large software programs lies. A mismatch of interaction patterns, such as calling methods on objects versus using OS signaling primitives versus using queues plus signaling, leads to unreadable code, where the runtime behavior changes drastically between coordination approaches. More importantly, the error handling approaches are ill-defined and again vary drastically.

### Application Model

CCR is appropriate for an application model that separates components into pieces that can interact only through messages. Components in this model need means to coordinate between messages, deal with complex failure scenarios, and effectively deal with asynchronous programming. This application model is also very similar to how heterogeneous hardware is integrated and how network applications are built. Most software programs have the same needs, from traditional client PC programs to server applications, to applets running in the Web browser. The software needs to coordinate user input, storage input/output and UI presentation. Although disguised in layers of mostly synchronous application code interfaces, asynchrony is inevitable since the devices operate at different speeds, have large differences in resources available, and we in general know how to use queues to isolate them.

The following sections introduce the CCR programming model and its implementation that addresses the above areas in an efficient, robust and extensible way.

Concurrency and Coordination Runtime (CCR) implementation has three main categories of functionality:

1. The **Port** and **PortSet** queueing primitives, see [CCR Ports and PortSets](mk:@MSITStore:C:\Users\ac\Microsoft%20Robotics%20Dev%20Studio%204\documentation\RDSUserGuide.chm::/CcrPortsAndPortSets.html). The Port class is a first in first out (FIFO) queue of items - an item can be any valid Common Language Runtime (CLR) type (only items of that type, either CLR base types or user-defined types, can be posted to the port). In most cases there is also a queue of receivers (user code "guarded" by an **Arbiter**, which means that the Arbiter controls the execution of tasks by filtering the incoming messages as appropriate).
2. The coordination primitives also called **Arbiters**, see [CCR Coordination Primitives](mk:@MSITStore:C:\Users\ac\Microsoft%20Robotics%20Dev%20Studio%204\documentation\RDSUserGuide.chm::/CcrCoordinationPrimitives.html). These are classes that execute user code, often a delegate to some method (which might be anonymous), when certain conditions are met. The primitives can be nested and can also be extended by the programmer.
3. The **Dispatcher**, **DispatcherQueue** and **Task** primitives, see [CCR Task Scheduling](mk:@MSITStore:C:\Users\ac\Microsoft%20Robotics%20Dev%20Studio%204\documentation\RDSUserGuide.chm::/CcrTaskScheduling.html). The CCR isolates the scheduling and load balancing logic from the rest of the implementation by using these classes to abstract how user tasks execute, what they contain, and on what resources they will run. CCR avoids a single, process wide execution resource, like the Common Language Runtime thread pool and instead allows the programmer to have multiple, completely isolated pools of OS threads that abstract any notion of threading behind them. (Unlike the CLR, the CCR thread pools are fixed in size at the time of creation and threads are not dynamically added or removed which makes CCR more efficient at managing threads). In the most common case, hundreds of software components can share just a single Dispatcher resource which can load balance millions of tasks across some small number of OS threads. The DispatcherQueue class is the only way to interact with the Dispatcher class, and multiple queues per dispatcher allow for a fair scheduling policy.

The Concurrency and Coordination Runtime (CCR) **Port** is the most common primitive and is used as the point of interaction between any two components. The port is implemented as the combination of two interfaces: **IPortReceive** and **IPort**. The interfaces logically separates methods that add items to the port, and methods that retrieve items, or attach code that removes items and executes asynchronously. The interfaces are not strongly typed. They work with instances of type object, but the default implementation, the **Port<>** class, has one generic type argument.

### 1. Ports and port sets (examples 1 – 6)

|  |  |
| --- | --- |
|  | The **<>** notation next to a class indicates its a generic class (similar to templates in C++). For example **Port<int>** is a port implementation with one Common Language Runtime (CLR) type argument of **System.Int32**. |

#### Posting Items

Adding an item to the port is an asynchronous operation. When there are no arbiters attached to the port, the item is added to a queue. If there are arbiters present, each one will be called to evaluate the item and decide if a task should be created and scheduled for execution. This evaluation is a fast, non-blocking operation so the Post method returns control to the caller as soon as possible. This is the source of the asynchronous behavior in the CCR: If certain conditions are met. Programs post items quickly to ports, which can then cause tasks to get scheduled in some thread other than the current one.

Example 1C#

// Create port that accepts instances of System.Int32

Port<int> portInt = new Port<int>();

// Add the number 10 to the port

portInt.Post(10);

// Display number of items to the console

Console.WriteLine(portInt.ItemCount);

In Example 1, a port typed is created to accept only integers. Then one item is added, the number 10, to the port. The port **ItemCount property** is checked, which should read 1.

###### Q1: what are the fields in the portInt variable? what are the methods of the portInt variable?

#### Retrieving Items

There are two scenarios when retrieving items from a port:

1. The port is used as a passive queue: no tasks are scheduled when items are posted. In this scenario items can be retrieved by calling the Test method. The method will never block. If no item is present it will return false and set the out parameter to null. This scenario is useful when there is some other mechanism to execute code due to some event, and the user simply wants to use the ports as efficient first in first out queues. This also provides the flexibility of later attaching a receiver to the port, that can asynchronously execute tasks for any item already present
2. The port is used as an active queue: tasks are scheduled for execution when items are posted, due to one or more arbiter being registered with the port. This is the most common case of using CCR ports. It is the primary source of concurrency since any user code that gets scheduled due to an item being posted, can potentially execute in parallel for every item posted

**Example 2**C#

// Create port that accepts instances of System.Int32

var portInt = new Port<int>();

// Add the number 10 to the port

portInt.Post(10);

// Display number of items to the console

Console.WriteLine(portInt.ItemCount);

// retrieve the item using Test

int item;

var hasItem = portInt.Test(out item);

if (hasItem)

{

Console.WriteLine("Found item in port:" + item);

}

portInt.Post(11);

// alternative to using Test is just assignment of port to variable using

// implicit operator

var nextItem = portInt;

Console.WriteLine("Found item in port:" + nextItem);

Example 2, an extension of Example 1, uses the Test method to retrieve the item posted. The item is removed in a thread-safe manner and **ItemCount** should read zero.

###### Q2: Find the C# "var" keyword in Microsoft's MSDN website. What does the "var" keyword do? What are the advantages and disadvantages to using "var"? What is the alternative? Be specific.

**Example 3**

// Create port that accepts instances of System.Int32

var portInt = new Port<int>();

// Add the number 10 to the port

portInt.Post(10);

// Display number of items to the console

Console.WriteLine(portInt.ItemCount);

// create dispatcher and dispatcher queue for scheduling tasks

Dispatcher dispatcher = new Dispatcher();

DispatcherQueue taskQueue = new DispatcherQueue("sample queue", dispatcher);

// retrieve the item by attaching a one time receiver

Arbiter.Activate(

taskQueue,

portInt.Receive(delegate (int item) // anonymous method

{

// this code executes in parallel with the method that

// activated it

Console.WriteLine("Received item:" + item);

}

));

// any code below runs in parallel with delegate

Example 3 follows the same steps as Example 1, but instead of using the Test method, the **Arbiter.Activate** method is used, to register a simple receiver arbiter to the port. The receiver is associated with a user delegate, in this case an anonymous method defined inline. The delegate executes in one of the dispatcher threads associated with the **DispatcherQueue** instance supplied. The Dispatcher and **DispatcherQueue** classes are not the focus here, and are described in more detail in the Scheduling Tasks section. Note the delegate always runs in parallel with the main body of this example. In this example, after the delegate runs, the receiver is automatically removed from the port. More information on receivers and arbiters is presented in a later section.

###### Q3: Read the following document about CCR coordination primitives:

###### <http://msdn.microsoft.com/en-us/library/bb648749.aspx>

###### what is the most primitive way to run a method on a CCR thread? What are the two scenarios that define CCR coordination primitives?

#### Inspecting Port State

The following methods and properties on the **Port<>** class are useful for inspecting its state at runtime, either for debugging or for regular code logic

* **ToString** - This method overrides the default **ToString()** implementation and will produce, in human readable form, the number of items in the port and the hierarchy of the receivers associated with the port.
* **ItemCount** - Number of items currently queued. Note that anytime one or more persisted arbiters (they stay attached even after consuming one item) is registered with the port, this count will be zero. Most arbiters cause items to be immediately converted to Tasks, scheduled for execution, so the items never actually show up in the port item queue.

### Port Sets

Since the **Port<>** class only takes one generic type argument, its often convenient to associate multiple, independent instances of the **Port<>** class, under a single container that itself can be treated as one entity. The **PortSet<>** generic class allows this grouping and is the most common way to define the interface to a CCR software component: Instead of having a set of public methods, CCR components expose only a strongly typed **PortSet**, with **N** independent **Port<>** instances underneath. The component can now coordinate how different message types execute in relation to each other. Further, the code always runs concurrently and independently from the code that posts the messages. This programming model is very familiar to authors of web services, traditional server processes, or **kernel mode I/O processors**.

#### Constructing a PortSet instance

There are two ways to create a new instance of PortSet:

1. Use the generic type arguments to define at compile time, the number and type of **Port<>** instances the **PortSet** will support. The CCR provides implementations of the **PortSet** of up to twenty generic type arguments on the desktop CLR, and up to eight generic arguments on the .NET Compact Framework, due to a JIT limitation. Using this approach gives you the best type safety and best runtime performance when posting. However it limits the number of types the port set can support.
2. Use the initialization constructor that takes a parameter list of type arguments. The user can supply an arbitrary number of types, and the **Port<>** instances will be created at runtime. Some methods, like the Post for each message type is not available (**PostUnknownType** or **TryPostUnknownType** must be used) and there is a minor performance hit since the item type has to be checked and compared at runtime, against a table of valid types.

**Example 4**C#

// Create a PortSet using generic type arguments

var genericPortSet = new PortSet<int, string, double>();

genericPortSet.Post(10);

genericPortSet.Post("hello");

genericPortSet.Post(3.14159);

// Create a runtime PortSet, using the initialization

// constructor to supply an array of types

PortSet runtimePortSet = new PortSet(

typeof(int),

typeof(string),

typeof(double)

);

runtimePortSet.PostUnknownType(10);

runtimePortSet.PostUnknownType("hello");

runtimePortSet.PostUnknownType(3.14159);

Example 4 shows how to create a **PortSet** with three different types, using the generic type arguments, and then using the type array at runtime. The strongly typed **Post** methods can be added to a class that derives from the non generic PortSet, providing the same compile time safety as the generic **PortSet<>**.

###### **Q4: concisely, what is the difference between the two port sets defined in this example?**

**Example 5**C#

/// <summary>

/// PortSet that accepts items of int, string, double

/// </summary>

public class CcrConsolePort : PortSet<int, string, double>

{

}

/// <summary>

/// Simple example of a CCR component that uses a PortSet to abstract

/// its API for message passing

/// </summary>

public class CcrConsoleService

{

CcrConsolePort \_mainPort;

DispatcherQueue \_taskQueue;

/// <summary>

/// Creates an instance of the service class, returning only a PortSet

/// instance for communication with the service

/// </summary>

/// <param name="taskQueue"></param>

/// <returns></returns>

public static CcrConsolePort Create(DispatcherQueue taskQueue)

{

var console = new CcrConsoleService(taskQueue);

console.Initialize();

return console.\_mainPort;

}

/// <summary>

/// Initialization constructor

/// </summary>

/// <param name="taskQueue">DispatcherQueue instance used for scheduling</param>

private CcrConsoleService(DispatcherQueue taskQueue)

{

// create PortSet instance used by external callers to post items

\_mainPort = new CcrConsolePort();

// cache dispatcher queue used to schedule tasks

\_taskQueue = taskQueue;

}

private void Initialize()

{

// Activate three persisted receivers (single item arbiters)

// that will run concurrently to each other,

// one for each item/message type

Arbiter.Activate(\_taskQueue,

Arbiter.Receive<int>(true, \_mainPort, IntWriteLineHandler),

Arbiter.Receive<string>(true, \_mainPort, StringWriteLineHandler),

Arbiter.Receive<double>(true, \_mainPort, DoubleWriteLineHandler)

);

}

void IntWriteLineHandler(int item)

{

Console.WriteLine("Received integer:" + item);

}

void StringWriteLineHandler(string item)

{

Console.WriteLine("Received string:" + item);

}

void DoubleWriteLineHandler(double item)

{

Console.WriteLine("Received double:" + item);

}

}

Example 5 defines a simple class, **CcrConsolePort**, that derives from a **PortSet** with three type arguments. This makes any subsequent use of the port set more readable since the generic type definitions don't have to be repeated. The class **CcrConsoleService**, implements a common pattern for CCR components. It has a static routine that creates the instance object and returns the private **PortSet** instance for communicating with it. It then activates a handler for each message type in the **PortSet**. Each handler activation is concurrent, assuming multiple threads are available for the dispatcher associated with the dispatcher queue.

#### Type Safety

The **PortSet** class allows for enumeration of all the port instances and item types it supports. The generic **PortSet** also provides convenient methods that automatically invoke the correct **Port<>** instance plus implicit conversion operations that can cast a **PortSet<>** instance to a **Port<>** instance, automatically. If the type argument is valid for that **PortSet**. This is useful when registering receivers with one of the ports in the **PortSet**, using just the type argument.

###### Q5: Write code to invoke each of the type handlers. Set breakpoints in the debugger on each handler. DEMONSTRATE

**Example 6**C#

var portSet = new PortSet<int, string, double>();

// the following statement compiles because of the implicit assignment operators

// that "extract" the instance of Port<int> from the PortSet

var portInt = portSet;

// the implicit assignment operator is used below to "extract" the Port<int>

// instance so the int receiver can be registered

Arbiter.Activate(\_taskQueue,

Arbiter.Receive<int>(true, portSet, item => Console.WriteLine(item))

);

Example 6 demonstrates the use of the implicit operator in two use cases:

1. Assigning the correct instance of a **Port<>** within a **PortSet**, to another **Port<>** variable
2. Extracting the correct instance of a **Port<>** so it can be used to register an arbiter

###### q6: Invoke the port handler for example 6. How would you explicitly register the int receiver?

### 2. CCR coordination primitives (examples 7 – 15)

Asynchronous programming is hard because there is no simple method to coordinate between multiple operations, deal with partial failure (one of many operations fail but others succeed) and also define execution behavior of asynchronous callbacks, so they don't violate some concurrency constraint. For example, they don't attempt to do something in parallel. The Concurrency and Coordination Runtime (CCR) enables and promotes concurrency by providing ways to express what coordination should happen. Plus, enforce the high level constraints between different code segments, all run due to some messages being received on ports.

At the most primitive level, you can run a method on a CCR thread using **Spawn(Handler)** which is defined in CcrServiceBase. There are a couple of additional overloads of Spawn that accept 1, 2 or 3 parameters to pass to the handler, e.g. **Spawn<T0>(T0, Handler<T0>)** which passes a parameter of type **T0**. You can call Spawn multiple times, but eventually you will use all of the available threads and they are free-running, i.e. there is no coordination amongst them apart from what they might do themselves.

It’s important to realize the relationship between asynchronous behavior and concurrency: The loose coupling, fast initiation of work, and consistent use of queues for interaction, promotes software design that scales and has well-defined dependencies. So if the drawbacks mentioned above can be addressed, it is an appropriate model for software components.

The coordination primitives provided by the CCR can be classified based on two primary use scenarios:

1. Coordination of inbound requests, for long lived service-oriented components. A common example is a web service listening for HTTP requests on some network port, using a CCR port to post all inbound requests and attaching handlers that wake up and serve each request independent of each other. In addition, it uses some of the advanced primitives to guarantee that some handlers never run when others handlers are active.
2. Coordination of responses from one or more outstanding requests, with multiple possible return types per response. One example is waiting for success or failure, on a **PortSet** associated with a pending request. When the request completes, a success item or a failure item will be posted, and the appropriate code should execute. Another example is scattering multiple requests at once. Then collecting all the responses using a single primitive, not caring what order the responses arrive across a single or multiple response ports.

Each arbiter will be described briefly, followed by a more detailed explanation in the context of the above scenarios.

### Arbiter Static Class

The arbiter static class provides helper routines for creating instances of all CCR arbiter classes, in a discoverable, type-safe manner. All methods described below are members of this class. The arbiter static methods are not an exhaustive list of all the possible ways the CCR arbiters can be constructed. For more advanced configurations, each arbiter class can be constructed directly using the appropriate constructor parameters. The following list shows which CCR classes are created when invoking some of the most common arbiter class methods (this is not an exhaustive list):

* **Arbiter.FromTask** -> Creates instance of **Task**
* **Arbiter.Choice** -> Creates instance of **Choice**
* **Arbiter.Receive** -> Creates instance of **Receiver**
* **Arbiter.Interleave** -> Creates instance of **Interleave**
* **Arbiter.JoinedReceive** ->Creates instance of **JoinReceiver**
* **Arbiter.MultipleItemReceive** -> Creates instance **JoinSinglePortReceiver**

The above classes are described in the reference documentation and should be used for advanced scenarios when additional parameters/configurations are necessary.

### Port Extension Methods

A more concise alternative to the static methods in the Arbiter class is available through the port extension methods. Using the new C# 3.0 extension method support, coordination primitives can be created by using the port instance. Most examples in the guide use the port extensions to create receivers, joins, etc. For example, **get.ResponsePort.Choice()** is an easy way to handle the response to a **Get** request to a DSS service. (DSS services expose PortSets that contain a port for the requested information and one for a **Fault**, so Choice is used to handle whichever message type is sent back. It effectively waits on the two ports at the same time and executes the corresponding delegate for whichever port receives a message).

### Single Item Receiver

A single item receiver associates a user delegate, that takes a single parameter of type **T**, with an instance of **Port<T>**. If the **persist** option is true, the receiver will execute an instance of the user delegate for every item posted. If the **persist** option is false, the receiver will only execute the user delegate for one item and then un-register from the port.

|  |  |
| --- | --- |
|  | If items are already queued on the **Port<T>** instance, the receiver will still execute them, providing a reliable way to "catch-up" to queued items, and make the user code agnostic on when the items where actually posted. |

**Example 7**C#

var port = new Port<int>();

Arbiter.Activate(\_taskQueue,

Arbiter.Receive(

true,

port,

item => Console.WriteLine(item)

)

);

// post item, so delegate executes

port.Post(5);

Example 7 shows how to create a **Port<int>** instance and then activate a single item receiver. This receiver will execute the user delegate every time an item is posted on the port. Notice that the receiver is persisted and it will be active on the port until the port instance is garbage collected.

Also in the example notice the abbreviated syntax for an anonymous delegate. The statement:

item => Console.WriteLine(item)

creates an anonymous method with one parameter (called item) that executes a single line of code to print the value on the console. If you are not an experienced C# programmer then this syntax might be new to you. An alternative using the older anonymous delegate style of syntax is:

delegate(int item)

{

Console.WriteLine(item);

}

###### Q7: implement the older anonymous delegate style. Is there an advantage to either approach?

**Example 8**

C#

// alternate version that explicitly constructs a Receiver by passing

// Arbiter class factory methods

var persistedReceiver = new Receiver<int>(

true, // persisted

port,

null, // no predicate

new Task<int>(item => Console.WriteLine(item)) // task to execute

);

Arbiter.Activate(\_taskQueue, persistedReceiver);

Example 8 has exactly the same effect at runtime as example 7, but shows that the **Arbiter.Receive()** method is really just a thin wrapper around the constructor of the Receiver arbiter.

### Choice Arbiter

The choice arbiter only executes one of its branches, and then, atomically (in one step that cant be interrupted) removes all other nested arbiters from their ports. This guarantees that only one branch of the choice will ever run and is a common way to express branching behavior, deal with responses that have success/failure, or guard against race conditions.

**Example 9**

C#

// create a simple service listening on a port

ServicePort servicePort = SimpleService.Create(\_taskQueue);

// create request

GetState get = new GetState();

// post request

servicePort.Post(get);

// use the extension method on the PortSet that creates a choice

// given two types found on one PortSet. This a common use of

// Choice to deal with responses that have success or failure

Arbiter.Activate(\_taskQueue,

get.ResponsePort.Choice(

s => Console.WriteLine(s), // delegate for success

ex => Console.WriteLine(ex) // delegate for failure

));

Example 9 shows one common use of the choice arbiter to execute two different delegates, based on messages received on a **PortSet**. Note that the choice class can take an arbitrary number of receivers, and coordinate across them, not just two. The Choice extension method on **PortSet** is a concise alternative to creating a choice arbiter, and then creating two receiver arbiters, one for each delegate.

|  |  |
| --- | --- |
|  | The choice arbiter is an example of a parent arbiter: other arbiters, such as single item receivers or joins, can be nested under a choice. The arbiter design allows for a hierarchy of arbiters, invoking each arbiter in the hierarchy in the correct order, before determining if a user handler should execute. This allows you to express complex coordination with just a few lines of code. |

###### Q9.1: What happens when this code runs? Why?

###### Q9.2: create a new UpdateState object. Post this object to servicePort so that it runs the delegate for success in the choice arbiter.

###### Hint : The class definitions for Example 9 can be found in Example 14

### Joins and Multiple Item Receivers

Multiple item receiver arbiters come in two categories:

1. Also known as joins, or **WaitForMultiple** in OS literature, they are receivers that attempt to receive from one or more ports. If one of the attempts fail, they post any items back and wait to try again when the right conditions are met. This two phase logic provides a type safe and deadlock free mechanism. It can be used to guarantee atomic access to multiple resources, without the fear of deadlock, since the order the items are received is not important. The number of items and ports can be specified at runtime or be fixed at compile time. The fact that the number of items in the join can be specified at runtime is an important extension the CCR provides over other forms of typed joins.
2. Receivers that eagerly remove items from each port participating in the receive, and when the total item count is satisfied, execute the user delegate. This version is very fast but should not be used as a resource synchronization primitive. It is often used for gathering results for multiple pending requests, known as scatter/gather scenarios.

#### Joins

**Example 10**

C#

var portDouble = new Port<double>();

var portString = new Port<string>();

// activate a joined receiver that will execute only when one

// item is available in each port.

Arbiter.Activate(\_taskQueue,

portDouble.Join(

portString, // port to join with

(value, stringValue) => // delegate

{

value /= 2.0;

stringValue = value.ToString();

// post back updated values

portDouble.Post(value);

portString.Post(stringValue);

})

);

// post items. The order does not matter, which is what Join its power

portDouble.Post(3.14159);

portString.Post("0.1");

//after the last post the delegate above will execute

Example 10 demonstrates a simple static join, as specified at compile time with a fixed number of ports. A join receiver is activated across two ports, and then posts items to each port. The join logic then determines if it has everything it needs and schedules the delegate for execution.

###### Q10.1: modify the example so that the updated values are printed from the receive handler.

###### Q10.2: modify the example so that the port values are read immediately following the post. What happens and why? How can you make the operation correct?

**Example 11**

C#

var portInt = new Port<int>();

var portDouble = new Port<double>();

var portString = new Port<string>();

// activate a joined receiver that will execute only when one

// item is available in each port.

Arbiter.Activate(\_taskQueue,

portDouble.Join(

portString, // second port to listen

(value, stringValue) =>

{

value /= 2.0;

stringValue = value.ToString();

// post back updated values

portDouble.Post(value);

portString.Post(stringValue);

})

);

// activate a second joined receiver that also listens on portDouble

// and on a new port, portInt. Because the two joins share a common port

// between them (portDouble), there is contention when items are posted on

// that port

Arbiter.Activate(\_taskQueue,

portDouble.Join(

portInt, // second port to listen

(value, intValue) =>

{

value /= 2.0;

intValue = (int)value;

// post back updated values

portDouble.Post(value);

portInt.Post(intValue);

})

);

// post items.

portString.Post("0.1");

portInt.Post(128);

// when the double is posted there will be a race

// between the two joins to determine who will execute first

// The delegate that executes first will then post back a double,

// allowing the delegate that "lost", to execute.

portDouble.Post(3.14159);

Example 11 is a simple extension of the previous join example showing a simple case of join contention. Two independent delegates listen on the same port, plus some other ports not common between them. Because the join implementation is two phase, there is no guarantee both will run, as soon as the value extracted from the shared port, is posted back. The order they run does not matter, so races will not affect the outcome. This basically shows how a traditional locking problem, across multiple resources, can become just a scheduling dependency, resolved by the CCR. Messages are both the resource being guarded from multiple concurrent access, and the signal that triggers the execution of the code that requires it.

|  |  |
| --- | --- |
|  | Using joins this way is a good alternative to using nested locks, but it is still a very error-prone way to program resource access. The Interleave primitive described in a later section, is a much simpler, less error-prone, and faster alternative. Q11.1: Modify your arbiters to output the modified values. What is the output? Why?Q11.2: How could the output have been different?Q11.3: How could you control the order of execution explicitly? |

**Example 12**

C#

int itemCount = 10;

var portDouble = new Port<double>();

// post N items to a port

for (int i = 0; i < itemCount; i++)

{

portDouble.Post(i \* 3.14159);

}

// activate a Join that

// waits for N items on the same port

Arbiter.Activate(\_taskQueue,

portDouble.Join(

itemCount,

items =>

{

foreach (double d in items)

{

Console.WriteLine(d);

}

}

)

);

Example 12 shows a simple case of a dynamic join: The number of items is known only at runtime, stored in variable **itemCount**. They are all read from one port. The example uses a version of join that executes a handler when N items are received from one port. The **Join()** extension method on the Port class is an an alternative to the **Arbiter.JoinedReceive()** and **Arbiter.MultipleItemReceive()** static methods.

###### Q12: Modify the join to use "itemCount+1". What happens? Why?

#### Multiple Item receivers

Multiple item receivers are appropriate when no contention is expected on the ports. They can be used to aggregate responses from multiple outstanding requests.

**Example 13**C#

// create a simple service listening on a port

var servicePort = SimpleService.Create(\_taskQueue);

// shared response port

var responsePort = new PortSet<string, Exception>();

// number of requests

int requestCount = 10;

// scatter phase: Send N requests as fast as possible

for (int i = 0; i < requestCount; i++)

{

// create request

GetState get = new GetState();

// set response port to shared port

get.ResponsePort = responsePort;

// post request

servicePort.Post(get);

}

// gather phase:

// activate a multiple item receiver that waits for a total

// of N responses, across the ports in the PortSet.

// The service could respond with K failures and M successes (K+M == N)

Arbiter.Activate(\_taskQueue,

responsePort.MultipleItemReceive(

requestCount, // total responses expected

(successes, failures) => Console.WriteLine("Total received:" + successes.Count + failures.Count)

)

);

Example 13 shows a common case for dealing with multiple pending asynchronous operations, using a single delegate to gather the results. Assuming for any N operations, K can fail, M can succeed, and K+M = N - the CCR **MultipleItemReceiver** gives a concise way to gather all the results, arriving in any order and in any combination across the types. A single delegate will be called, with two collections, containing the K failures and M successes. The **MutipleItemReceive** extension method can be used for two discrete types but the underlying **MultipleItemGather** CCR arbiter can work with an arbitrary number of types.

Note that the **MultipleItemReceiver** in Example 13 must be activated or it will have no effect. The code will compile, but the receiver will not be executed. Using **yield return** does an implicit activation, so there is no need to use **Arbiter.Activate**. However, **yield return** can only be used inside an Iterator.

###### Q13.1: What are the "successes" and "failures" parameters in the delegate?

###### Q13.2: Modify the example so that all responses are successful.

### Coordination for service-oriented components

#### Persisted Single Item Receivers

CCR was motivated from the beginning as the runtime capable of efficiently executing components that listen on some queues for messages, and activate handlers to process inbound messages. The simplest case is to use the receiver arbiter, in persisted mode, to listen on a port and activate a handler whenever an item is posted.

**Example 14**C#

/// <summary>

/// Base type for all service messages. Defines a response PortSet used

/// by all message types.

/// </summary>

public class ServiceOperation

{

public PortSet<string, Exception> ResponsePort = new PortSet<string, Exception>();

}

public class Stop : ServiceOperation

{

}

public class UpdateState : ServiceOperation

{

public string State;

}

public class GetState : ServiceOperation

{

}

/// <summary>

/// PortSet that defines which messages the services listens to

/// </summary>

public class ServicePort : PortSet<Stop, UpdateState, GetState>

{

}

/// <summary>

/// Simple example of a CCR component that uses a PortSet to abstract

/// its API for message passing

/// </summary>

public class SimpleService

{

ServicePort \_mainPort;

DispatcherQueue \_taskQueue;

string \_state;

public static ServicePort Create(DispatcherQueue taskQueue)

{

var service = new SimpleService(taskQueue);

service.Initialize();

return service.\_mainPort;

}

private void Initialize()

{

// using the supplied taskQueue for scheduling, activate three

// persisted receivers, that will run concurrently to each other,

// one for each item type

Arbiter.Activate(\_taskQueue,

Arbiter.Receive<UpdateState>(true, \_mainPort, UpdateHandler),

Arbiter.Receive<GetState>(true, \_mainPort, GetStateHandler)

);

}

private SimpleService(DispatcherQueue taskQueue)

{

// create PortSet instance used by external callers to post items

\_mainPort = new ServicePort();

// cache dispatcher queue used to schedule tasks

\_taskQueue = taskQueue;

}

void GetStateHandler(GetState get)

{

if (\_state == null)

{

// To demonstrate a failure response,

// when state is null will post an exception

get.ResponsePort.Post(new InvalidOperationException());

return;

}

// return the state as a message on the response port

get.ResponsePort.Post(\_state);

}

void UpdateHandler(UpdateState update)

{

// update state from field in the message

\_state = update.State;

// as success result, post the state itself

update.ResponsePort.Post(\_state);

}

}

Example 14 shows a class implementing the common CCR pattern for a software component:

* Definitions of message types used to interact with the component.
* Definition of a **PortSet** derived class that accepts the message types defined. Its not necessary to derive from **PortSet**, but its a convenient way to reuse a **PortSet** with a particular number of types.
* A static **Create** method that initializes an instance of the component and returns a **PortSet** instance used to communicate with the component instance.
* A private **Initialize** method that attaches some arbiters on the public **PortSet** external code will use to talk to the service.

If no concurrency constraints exist between the different handlers, simple, persisted single item receivers can be used.

###### Q14: HINT - this code is needed for the last 5 examples. Why do you think the SDK hid it back here?

#### Interleave Arbiter

For non trivial components that listen on ports, often a private resource is used that should be carefully protected from concurrent access. A data structure stored internally requiring multiple updates that must be treated atomically is one case. Another scenario is a component implementing a complex multi-step process, that cannot be preempted when certain external requests arrive. The CCR helps you think only about implementing the complex process, and takes care of queueing requests and handler activations, until the process is complete. You use the interleave arbiter to declare what protection segments of code require.

For programmers familiar with the reader/writer lock primitive in thread programming, the interleave arbiter is a similar concept. It’s a writer biased reader/writer. But, instead of locking a specific object, sections of code are protected from each other. Avoiding contention on a lock, interleave uses internal queues to create scheduling dependencies. Plus, it manages execution so tasks that can run concurrently, do, and tasks that run exclusively, first wait for all other tasks to complete.

**Example 15**

C#

/// <summary>

/// Simple example of a CCR component that uses a PortSet to abstract

/// its API for message passing

/// </summary>

public class ServiceWithInterleave

{

ServicePort \_mainPort;

DispatcherQueue \_taskQueue;

string \_state;

public static ServicePort Create(DispatcherQueue taskQueue)

{

var service = new ServiceWithInterleave(taskQueue);

service.Initialize();

return service.\_mainPort;

}

private void Initialize()

{

// activate an Interleave Arbiter to coordinate how the handlers of the service

// execute in relation to each other and to their own parallel activations

Arbiter.Activate(\_taskQueue,

Arbiter.Interleave(

new TeardownReceiverGroup(

// one time, atomic teardown

Arbiter.Receive<Stop>(false, \_mainPort, StopHandler)

),

new ExclusiveReceiverGroup(

// Persisted Update handler, only runs if no other handler running

Arbiter.Receive<UpdateState>(true, \_mainPort, UpdateHandler)

),

new ConcurrentReceiverGroup(

// Persisted Get handler, runs in parallel with all other activations of itself

// but never runs in parallel with Update or Stop

Arbiter.Receive<GetState>(true, \_mainPort, GetStateHandler)

))

);

}

private ServiceWithInterleave(DispatcherQueue taskQueue)

{

// create PortSet instance used by external callers to post items

\_mainPort = new ServicePort();

// cache dispatcher queue used to schedule tasks

\_taskQueue = taskQueue;

}

void GetStateHandler(GetState get)

{

if (\_state == null)

{

// when state is null will post an exception

get.ResponsePort.Post(new InvalidOperationException());

return;

}

// return the state as a message on the response port

get.ResponsePort.Post(\_state);

}

void UpdateHandler(UpdateState update)

{

// update state from field in the message

// Because the update requires a read, a merge of two strings

// and an update, this code needs to run un-interrupted by other updates.

// The Interleave Arbiter makes this guarantee since the UpdateHandler is in the

// ExclusiveReceiverGroup

\_state = update.State + \_state;

// as success result, post the state itself

update.ResponsePort.Post(\_state);

}

void StopHandler(Stop stop)

{

Console.WriteLine("Service stopping. No other handlers are running or will run after this");

}

}

Example 15 extends the SimpleService class to use an interleave arbiter to coordinate the receivers that execute the various handlers. Interleave is another example of a parent arbiter that can have various other receivers nested. The example shows how you can concisely state intent in terms of concurrency. Certain handlers can run independently, others can not. The CCR does not need to know what resource or multi step process needs exclusive access. It only needs to know what code handler to protect. The handlers are very simple in this example. However, in a later section, iterator handlers demonstrate how interleave can protect complex code that runs in multiple steps.

If you are writing a DSS service, then your service is a subclass of the **DsspServiceBase** class which automatically creates a **MainInterleave** for you. Furthermore, the service handlers are automatically added to this interleave (when you call base.Start) if you mark them with the **ServiceHandler** attribute. For more information on this see the DSS documentation.

###### Q15.1: Implement code to demonstrate the ServiceWithInterleave class.

###### HINT see example 9

###### Q15.2: Execute both the failure and success branches

###### Q15.3: How could you verify that the operations are atomic?

### 3. Task scheduling (example 16 – 17)

The third major component of the Concurrency and Coordination Runtime (CCR) is how tasks, generated when messages arrive on ports with active receivers, get load balanced among the execution resources of the machine. There are three important classes that implement or abstract scheduling in the CCR:

* The task class, including **ITask** interface, the Task and **IterativeTask** implementations. Only classes that implement **ITask** can be scheduled. Arbiters also implement **ITask** so they can be scheduled and properly activate.
* The DispatcherQueue class. **DispatcherQueue** is a first in first out (FIFO) queue of Tasks. Dispatcher queues can use the Common Language Runtime (CLR) thread pool for scheduling tasks (very uncommon) or an instance of a CCR dispatcher
* The Dispatcher class. The dispatcher manages OS threads and load balances tasks de-queued from one or more DispatcherQueue instances.

**Example 16**C#

var dispatcher = new Dispatcher(

0, // zero means use one thread per CPU, or 2 if only one CPU present

"sample dispatcher" // friendly name assgined to OS threads

);

var taskQueue = new DispatcherQueue(

"sample queue", // friendly name

dispatcher // dispatcher instance

);

var port = new Port<int>();

Arbiter.Activate(taskQueue,

Arbiter.Receive(

true,

port,

item => Console.WriteLine(item)

)

);

// post item, so delegate executes

port.Post(5);

Example 16 shows how a dispatcher and dispatcher queue are created and then used to schedule tasks. The following is a step-by-step description of the example:

1. An instance of a Dispatcher is created, using 0 as the number of threads. This makes the CCR choose a number of threads based on the number of CPU cores reported by the OS. The number of threads per CPU, used for the default, is controlled by the static property **ThreadsPerCpu** on the Dispatcher class.
2. An instance of a DispatcherQueue is created, supplying the Dispatcher instance we created in step 1. This attaches the DispatcherQueue.
3. An instance of **Port<int>** is created and used to post items and also attach a receiver with delegate.
4. The **Arbiter.Activate** method is called passing the instance of the DispatcherQueue created earlier, plus a new receiver arbiter with the port it needs to listen on, plus the delegate to execute when an item is posted on the port
5. An item of type **int** is posted on the port.

When an item is posted on a port with a receiver attached, the following happens within the port implementation:

1. A container is created for the value being posted. The container class, **IPortElement**, allows the CCR to queue items and also assign them to task instances, without caring about the item type.
2. The container instance is queued.
3. If the list of receivers is not null and there is at least one receiver, the port will call the **ReceiverTask.Evaluate** method so the receiver and its arbiter hierarchy can determine if the item can be consumed. In this example, the receiver will return true from evaluate and also create a new instance of **Task<int>** using the item and the user delegate as parameters.
4. The port logic calls **taskQueue.Enqueue** with the task returned from the evaluate method on the receiver. Note that when a receiver is first activated, it is associated with the dispatcher queue instance supplied in the **Arbiter.Activate** method

After step 4 above, the generated task instance is now dealt by the scheduling logic.

###### Q16: In the debugger, count the number of threads generated by your dispatcher. How many CPU cores on your machine? Why are these numbers related?

**Example 17**C#

// directly enqueue a task with an inlined method plus a parameter

taskQueue.Enqueue(

new Task<int>(5, item => Console.WriteLine(item))

);

Example 17 shows the equivalent code for scheduling the same delegate as in example 16 but without posting anything on a port. Creating a task instance explicitly is useful when data is available and code can be immediately executed to process it. The CCR does a similar task creation when a receiver is invoked in the context of a **Port.Post** call.

Once an item is queued in the dispatcher queue, the following happens:

1. The dispatcher queue signals the dispatcher instance it's associated with, that a new task is available for execution.
2. The dispatcher notifies one or more instances of its **TaskExecutionWorker** class. Each task execution worker manages one OS thread. When items are available for scheduling it puts the thread in an efficient sleep state, waiting for a signal from the dispatcher.
3. An instance of **TaskExecutionWorker** calls the **DispatcherQueue** test method to retrieve a task from the queue. If a task is available, and not already picked up by another worker, the worker calls **ITask.Execute.**
4. The **Task.Execute** invokes the delegate associated with the task, passing it one or more parameters associated with the task. In the example a single parameter with the value 5 is passed to the delegate that writes to the console.

###### Q17: How is this different from the approach in example 16? Which task would execute first if both are invoked sequentially?

### 3. CCR iterators (example 18 – 21)

Iterators are a C# 2.0 language feature that is used in a novel way by the Concurrency and Coordination Runtime (CCR). Instead of using delegates to nest asynchronous behavior, also known as callbacks, you can write code in a sequential fashion, yielding to CCR arbiters or other CCR tasks. Multi-step asynchronous logic can then be all written in one iterator method, vastly improving readability of the code. At the same time maintaining the asynchronous behavior, and scaling to millions of pending operations since no OS thread is blocked during a yield.

(Iterators can be used to step through objects like lists, arrays, etc. and on each call to the iterator the next item is returned until finally there are no more left. CCR iterators return a series of tasks to be executed, pausing between tasks as necessary. In this way, what appears to be sequential code is actually a series of small code snippets, or thunks, that are executed one after another, possibly on different threads. Note that a thread is never blocked during the execution of a CCR iterator. When an iterator needs to wait on a port, it returns the thread to the pool and another thread is scheduled later to execute the next piece of the code when a message arrives on the port.)

**Example 18**C#

void StartIterator()

{

// create an IterativeTask instance and schedule it

Arbiter.Activate(\_taskQueue,

Arbiter.FromIteratorHandler(IteratorExample)

);

}

/// <summary>

/// Iterator method scheduled by the CCR

/// </summary>

IEnumerator<ITask> IteratorExample()

{

// accumulator variable

int totalSum = 0;

var portInt = new Port<int>();

// using CCR iterators we can write traditional loops

// and still yield to asynchronous I/O !

for (int i = 0; i < 10; i++)

{

// post current iteration value to a port

portInt.Post(i);

// yield until the receive is satisifed. No thread is blocked

yield return portInt.Receive();

// retrieve value using simple assignment

totalSum += portInt;

}

Console.WriteLine("Total:" + totalSum);

}

Example 18 shows how the StartIterator method uses the arbiter class to create a task from an iterator delegate and then submits it for scheduling using **Arbiter.Activate**. The second method is the iterator method. The iterator method can use the yield return and yield break C# statements in its body to control execution. What distinguishes this method from more common C# methods is the return value:

IEnumerator<ITask> IteratorExample()

{

The return value indicates this is a C# iterator over CCR ITask instances and informs the compiler that the method body might contain yield statements.

yield return portInt.Receive(); {

The **yield return** statement above returns control to the CCR scheduler. It also returns an instance of the **ITask** interface, implemented by the **ReceiverTask** created when calling the **portInt.Receive** extension method. The CCR scheduler activates the returned task and also associates the iterator with the task. When the task completes execution, it can choose to progress the iterator to the next code statement after the yield. When you think about this code, you can think of the yield as a synchronization point: the iterator method will stop execution until the yield is satisfied.

Note that when code execution resumes at the line after the yield return, there is no guarantee that it will be the same CCR thread. Also, you cannot use the yield return statement in normal code, only within an Iterator. To terminate execution of the iterator early without "falling off the end" you can use the **yield break** statement. Iterators cannot return a value in the sense of a normal method, so they need to modify variables outside their scope, or preferably post a message to a port (passed in as a parameter) with the return value.

|  |  |
| --- | --- |
|  | Never yield to persisted receivers within an iterator. In the yield statement above, Receive creates a one time receiver. If the receiver is persisted, the yield is never considered satisfied, so the iterator will never progress to the next statement after the yield |

An alternative way to schedule an iterative task is to use **SpawnIterator()**. Just like Spawn() for conventional methods, there are three overloads of SpawnIterator that accept 1, 2, or 3 parameters that are passed to the iterative task, e.g. **SpawnIterator<T0>(T0, Handler<T0>)**. It is important to note that you cannot call an iterative task from normal code. Doing so will not generate any errors, but it will have no effect. If you want to execute an iterative task, you must use SpawnIterator() or execute it indirectly through an Arbiter. (See below).

###### Q18: HINT - you may want to mark the methods static to get them to execute or you could implement them within a class. What are the advantages or disadvantages of the two approaches?

### Implicit parameter passing through local variables

The power of using the CCR with the C# iterators comes from two other C# language features:

1. Anonymous methods: This feature is used in our examples to define the body of a handler as an in-lined delegate.
2. The compiler captures all local variables in the iterator method that are referenced inside the body of the anonymous method. This allows the delegate to use local variables, defined in the parent method. The delegates, that always run in some other thread, can communicate results back to the parent iterator, with no explicit parameter passing.

// retrieve value using simple assignment

totalSum += portInt;

The next statement after the yield is shown above. Using an implicit assignment operator, the code extracts the value from the port. Since the receive operation above was satisfied, and no handler was passed as an argument to **portInt.Receive()**, the item was kept in the port.

Console.WriteLine("Total:" + totalSum);

The line above is from the body of the iterator method **IteratorExample**, used in example 18. Its the last statement in the method and it implicitly marks the end of the iteration.

### Yielding to coordination primitives

**Example 19**

C#

void StartIterator2()

{

Port<string> portString = new Port<string>();

Arbiter.Activate(

\_taskQueue,

Arbiter.ReceiveWithIterator(false, portString, StringIteratorHandler)

);

}

IEnumerator<ITask> StringIteratorHandler(string item)

{

Console.WriteLine(item);

yield break;

}

Example 19 shows how to specify an iterator method as the outcome of a receive operation using **ReceiveWithIterator**. Until now the examples have been using traditional methods that execute when an arbiter is satisfied. The arbiter class contains methods that expect an iterator delegate for all the major coordination primitives, **JoinReceiver**, **MultipleItemGather**, **Choice**, **Receiver**, etc. This gives you the choice between regular methods or iterator methods for every CCR coordination primitive. The arbiter implementations work with instances of **ITask**, which hides the type of the user delegate, so they work with iterator or non-iterator methods.

###### Q19: compare the approaches in examples 19 and 18. Describe the difference (concisely).

**Example 20**C#

IEnumerator<ITask> IteratorWithChoice()

{

// create a service instance

ServicePort servicePort = ServiceWithInterleave.Create(\_taskQueue);

// send an update request

UpdateState updateRequest = new UpdateState();

updateRequest.State = "Iterator step 1";

servicePort.Post(updateRequest);

string result = null;

// wait for either outcome before continuing

yield return Arbiter.Choice(

updateRequest.ResponsePort,

response => result = response,

ex => Console.WriteLine(ex)

);

// if the failure branch of the choice executed, the result will be null

// and we will terminate the iteration

if (result == null)

yield break;

// print result from first request

Console.WriteLine("UpdateState response:" + result);

// now issue a get request

GetState get = new GetState();

servicePort.Post(get);

// wait for EITHER outcome

yield return Arbiter.Choice(

get.ResponsePort,

delegate(string response) { result = response; },

delegate(Exception ex) { Console.WriteLine(ex); }

);

// print result from second request

Console.WriteLine("GetState response:" + result);

}

Example 20 shows a common use case for iterators: When you require multiple asynchronous requests to a service, where the outcome of each request is success or failure. In real world examples there is usually some data dependency of request **N**, from the result of request **N-1**, which is why they are done in sequence. This is the ability to yield to the return value of **Arbiter.Choice**, which creates an instance of the Choice arbiter. Allowing the iterator to concisely handle multiple outcomes of an asynchronous I/O operation, all in-line. Note that the iterator method continues execution when either branch of the choice executes. It also uses the result stack variable to retrieve the result of the requests.

###### Q20: Force both branches of the choice to execute. DEMONSTRATE

### Nesting of iterators

When an iterator methods grows too large to be easily maintainable, it can be decomposed further to smaller iterator methods. The CCR scheduler can determine when an iterator has exhausted all its steps. This means the iterator reached a yield break or finished executing the last step of the iteration.

**Example 21**

C#

IEnumerator<ITask> ParentIteratorMethod()

{

Console.WriteLine("Yielding to another iterator that will execute N asynchronous steps");

yield return new IterativeTask<int>(10, ChildIteratorMethod);

Console.WriteLine("Child iterator completed");

}

IEnumerator<ITask> ChildIteratorMethod(int count)

{

Port<int> portInt = new Port<int>();

for (int i = 0; i < count; i++)

{

portInt.Post(i);

// short form of receive that leaves item in port

yield return portInt.Receive();

// implicit operator extracts item from port

int result = portInt;

}

}

Example 21 shows the nesting of iterators by simply creating a new IterativeTask and yielding to it:

* The parent iterator method yields execution to the new IterativeTask instance.
* CCR schedules the IterativeTask instance on the dispatcher queue instance used by the current iterator. **Arbiter.FromIteratorHandler** can also be used instead of explicitly creating an **IterativeTask** .
* The child iterator method executes in one of the dispatcher threads, and yields 10 times to a simple receive operation. A loop is used to showcase how iterators make looping around asynchronous operations very easy and readable. It also shows that the parent iterator can yield to arbitrarily complex child iterators, without knowing how many asynchronous steps they execute.

Note that you can yield to regular tasks, by using the **Arbiter.ExecuteToCompletion** method.

|  |  |
| --- | --- |
|  | If an exception is thrown in a handler executed in the context of a nested iterator, the parent iterator will still be called. Exceptions implicitly terminate iterators and the CCR properly disposes them. If causalities as present, the exception will be posted on the causality exception port, then the iterator will be disposed Q21: Modify the example above by outputting the thread id of each child execution in the iteration (Thread.CurrentThread.ManagedThreadId). Are all iterations run by the same thread? Why?Extra credit: modify execution so multiple threads are used to run the iteration |

### 4. CCR failure handling (example 22 – 24)

Traditional failure handling schemes follow these patterns:

1. For synchronous invocations of methods, the caller checks one or more return values from the method. The method uses the callers execution context, most often this is a thread, to run.
2. Using structured exception handling, the caller wraps synchronous method invocations in **try/catch/finally** statements and either relies exclusively on the **catch{}** block for error handling. It may also use a combination of the **catch{}** block plus explicit checks on the results from the method.
3. Transactions which rely on a variety of compensation implementations by the components being called, OS infrastructure and usually expensive mechanisms to flow context across threads.

All methods above can not easily apply to concurrent, asynchronous execution, especially the first two which are simply not available for asynchronous programming. Methods execute in arbitrary context, potentially in parallel with the caller. Unless there is a blocking operation to collect the result of the asynchronous operation, thus blocking a thread in the callers context. In this case, failure or success can not easily be determined. The asynchronous programming model in Common Language Runtime (CLR), using the Begin/End calls introduces additional complexity since failure handling happens through additional calls and exception handling in the single callback that executes regardless of the result of the operation. The code that started the asynchronous operations is not the place that failure is handled, making code hard to read.

The CCR addresses failure handling with two approaches:

1. Explicit or local failure handling using the **Choice** and **MultipleItemGather** arbiters. Combined with iterator support they provide a type safe, robust way to deal with failure since they force you to deal with the success and failure case in two different methods and then also have a common continuation. Examples 8,13 and 20 show explicit error handling using **Choice**, **MultipleItemGather** and **Choice** in an iterator, respectively.
2. Implicit or distributed failure handling, referred to as Causalities that allows for nesting and extends across multiple asynchronous operations. It shares in common with transactions the notion of a logical context or grouping of operations and extends it for a concurrent asynchronous environment. This mechanism can also be extended across machine boundaries

### Causalities

Causality refers to a sequence of operations, that can span multiple execution contexts, fork and join, creating a tree of execution logically rooted in one source, one cause. This logical grouping is called the causality context and implicitly flows from the sender of a message to the receiver. Any further messages sent by the code activated on the receiver, carry with them this causality and propagate it to any new receivers.

Causalities are an extension of the structured exception mechanism implemented for threads. They allow for nesting but also deal with multiple failures happening concurrently and with merges with other causalities, due to joins for example.

**Example 22**

C#

void SimpleCausalityExample()

{

Port<Exception> exceptionPort = new Port<Exception>();

// create a causality using the port instance

Causality exampleCausality = new Causality("root cause", exceptionPort);

// add causality to current thread

Dispatcher.AddCausality(exampleCausality);

// any unhandled exception from this point on, in this method or

// any delegate that executes due to messages from this method,

// will be posted on exceptionPort.

Port<int> portInt = new Port<int>();

Arbiter.Activate(\_taskQueue,

Arbiter.Receive(false, portInt, IntHandler)

);

// causalities flow when items are posted or Tasks are scheduled

portInt.Post(0);

// activate a handler on the exceptionPort

// This is the failure handler for the causality

Arbiter.Activate(\_taskQueue,

Arbiter.Receive(false, exceptionPort,

delegate(Exception ex)

{

// deal with failure here

Console.WriteLine(ex);

})

);

}

void IntHandler(int i)

{

// print active causalities

foreach (Causality c in Dispatcher.ActiveCausalities)

{

Console.WriteLine(c.Name);

}

// expect DivideByZeroException that CCR will redirect to the causality

int k = 10 / i;

}

Example 22 demonstrates a simple scenario where causalities help with error handling across asynchronous operations. The code performs the following key steps:

1. An instance of **Port<Exception>** is created to store any exceptions thrown within the causality context.
2. A new Causality instance is created using the exception port plus a friendly name.
3. The causality is added to the list of causalities the Dispatcher keeps for the current execution context, the OS thread.
4. An item is posted on a port. Because the post happens within the context of a causality, it will implicitly attach its causality to the item being posted. When a handler executes with this item, the causalities attached with the item will be added to the thread executing the receiver. Any exceptions in the receiver, will be posted on the exception port associated with the causality
5. A receiver is activated on the exception port, so any exception thrown and not handled by any asynchronous operations within the causality can be received.
6. The handler scheduled due to the item being posted on the portInt instance, throws a **DivideByZeroException**. The CCR scheduler redirects the exception to the exception port associated with the causality.

The example demonstrates that causalities are the most robust way to deal with failure. If you add additional asynchronous operations, that cause other handlers to execute in parallel, the failure handling is not modified. Further, any operations these handlers do, will also be covered within the scope of the original causality flowing exceptions all the way back to the exception port

###### Q22: Paste the output from this example into your solution

#### Nested Causalities

It is often appropriate to nest causalities, so different failure handlers compensate for failure at different levels. Similar to structured exceptions, but generalized to a concurrent setting. Causalities can nest, giving you the choice of either dealing with an exception in the inner causality, or posting it explicitly to any parent causality.

**Example 23**

C#

public void NestedCausalityExample()

{

Port<Exception> exceptionPort = new Port<Exception>();

Causality parentCausality = new Causality(

"Parent",

exceptionPort);

Dispatcher.AddCausality(parentCausality);

Port<int> portInt = new Port<int>();

Arbiter.Activate(\_taskQueue,

Arbiter.Receive(false, portInt, IntHandlerNestingLevelZero)

);

portInt.Post(0);

// activate a handler on the exceptionPort

// This is the failure handler for the causality

Arbiter.Activate(\_taskQueue,

Arbiter.Receive(false,

exceptionPort,

delegate(Exception ex)

{

// deal with failure here

Console.WriteLine("Parent:" + ex);

})

);

}

void IntHandlerNestingLevelZero(int i)

{

// print active causalities

foreach (Causality c in Dispatcher.ActiveCausalities)

{

Console.WriteLine("Before child is added: " + c.Name);

}

// create new child causality that will nest under existing causality

Port<Exception> exceptionPort = new Port<Exception>();

Causality childCausality = new Causality(

"Child",

exceptionPort);

Dispatcher.AddCausality(childCausality);

Arbiter.Activate(\_taskQueue,

Arbiter.Receive(false,

exceptionPort,

delegate(Exception ex)

{

// deal with failure here

Console.WriteLine("Child:" + ex);

})

);

// print active causalities

foreach (Causality c in Dispatcher.ActiveCausalities)

{

Console.WriteLine("After child is added: " + c.Name);

}

// attach a receiver and post to a port

Port<int> portInt = new Port<int>();

Arbiter.Activate(\_taskQueue,

Arbiter.Receive(false, portInt, IntHandlerNestingLevelOne)

);

portInt.Post(0);

}

void IntHandlerNestingLevelOne(int i)

{

throw new InvalidOperationException("Testing causality support. Child causality will catch this one");

}

##### Console output from example:

Before child is added: Parent

After child is added: Child

Child:System.InvalidOperationException: Testing causality support. Child causality will catch this one

at Examples.Examples.IntHandlerNestingLevelOne(Int32 i) in C:\mri\main\CCR\testsrc\UnitTests\ccruserguideexamples.cs:line 571

at Microsoft.Ccr.Core.Task`1.Execute() in C:\mri\main\CCR\src\Core\Templates\GeneratedFiles\Task\Task01.cs:line 301

at Microsoft.Ccr.Core.TaskExecutionWorker.ExecuteTaskHelper(ITask currentTask) in C:\mri\main\CCR\src\Core\scheduler\_roundrobin.cs:line 1476

at Microsoft.Ccr.Core.TaskExecutionWorker.ExecuteTask(ITask& currentTask, DispatcherQueue p) in C:\mri\main\CCR\src\Core\scheduler\_roundrobin.cs:line 1376

at Microsoft.Ccr.Core.TaskExecutionWorker.ExecutionLoop() in C:\mri\main\CCR\src\Core\scheduler\_roundrobin.cs:line 1307

The example above shows a nested asynchronous sequence:

1. Method **NestedCausalityExample** adds a causality and issues post to a port with a receiver.
2. Method **IntHandlerNestingLevelZero** executes asynchronously and in parallel with method **NestedCausalityExample**, adding a new causality that nests under the causality that flowed from step 1.
3. Method **IntHandlerNestingLevelZero** again creates a new port with a different receiver and posts a message.
4. Method **IntHandlerNestingLevelOne** executes, indirectly caused from the original post in method **NestedCausalityExample**, and with two active causalities. The bottom of the stack, is the child causality created by method **IntHandlerNestingLevelZero**.
5. Exception is thrown in method **IntHandlerNestingLevelOne**, which is handled by the child causality handler, since it "hides" exceptions from its parent.

Using the static property **Dispatcher.ActiveCausalities**, a causality exception handler can explicitly post exceptions to its parent causalities, propagating failure selectively to higher levels. The example uses this collection to print out all the active causalities at the top of method **IntHandlerNestingLevelZero**.

#### Joins and Causalities

A unique feature of the CCR causality implementation is that it can combine causalities that come from two different executions paths, but "meet" on a handler that executed due to a join or multiple item receive being satisfied. In this scenario the CCR does not nest causalities, but adds them as peers in the thread handler with the joined items. If an exception is thrown, it propagates concurrently along two independent causality stacks, one for each item passed in the handler.

**Example 24**

C#

public void JoinedCausalityExample()

{

Port<int> intPort = new Port<int>();

Port<int> leftPort = new Port<int>();

Port<string> rightPort = new Port<string>();

Port<Exception> leftExceptionPort = new Port<Exception>();

Port<Exception> rightExceptionPort = new Port<Exception>();

// post twice so two handlers run

intPort.Post(0);

intPort.Post(1);

// activate two handlers that will execute concurrently and create

// two different parallel causalities

Arbiter.Activate(\_taskQueue,

Arbiter.Receive(false, intPort,

delegate(int i)

{

Causality leftCausality = new Causality("left", leftExceptionPort);

Dispatcher.AddCausality(leftCausality);

// post item on leftPort under the context of the left causality

leftPort.Post(i);

})

);

Arbiter.Activate(\_taskQueue,

Arbiter.Receive(false, intPort,

delegate(int i)

{

Causality rightCausality = new Causality("right", rightExceptionPort);

Dispatcher.AddCausality(rightCausality);

// post item on rightPort under the context of the right causality

rightPort.Post(i.ToString());

})

);

// activate one join receiver that executes when items are available on

// both leftPort and rightPort

Arbiter.Activate(\_taskQueue,

Arbiter.JoinedReceive<int, string>(false, leftPort, rightPort,

delegate(int i, string s)

{

throw new InvalidOperationException("This exception will propagate to two peer causalities");

})

);

// activate a handler on the exceptionPort

// This is the failure handler for the causality

Arbiter.Activate(\_taskQueue,

Arbiter.Receive(false, leftExceptionPort,

delegate(Exception ex)

{

// deal with failure here

Console.WriteLine("Left causality: " + ex);

})

);

// activate a handler on the exceptionPort

// This is the failure handler for the causality

Arbiter.Activate(\_taskQueue,

Arbiter.Receive(false, rightExceptionPort,

delegate(Exception ex)

{

// deal with failure here

Console.WriteLine("Right causality: " + ex);

})

);

}

##### Console output from example.

Console

Left causality: System.InvalidOperationException: This exception will propagate to two peer causalities

Right causality: System.InvalidOperationException: This exception will propagate to two peer causalities

The example above demonstrates how anonymous methods are used heavily to keep all the logic within one method, for readability. The example demonstrates:

* Two independent handlers create two causalities, named left and right
* Post one message each to two different ports
* A third handler executes due to a join being satisfied across these two ports, **leftPort** and **rightPort**
* The join handler throws an exception in the context of both causalities

The key point is that when the join handler executes and throws an exception, two peer causalities are active and thus both independently get the exception posted on their respective exception ports.

|  |  |
| --- | --- |
|  | Since causalities use regular CCR ports to receive exceptions, you can use CCR coordination primitives to combine exception handling across multiple causalities. Joins, interleave and choice are all appropriate and powerful ways to combine failure handling across concurrent, multi-tiered asynchronous operations |