

2

Concurrency and Coordination Runtime (CCR)

Microsoft Robotics Developers Studio (MRDS) is built on two basic components: the Concurrency and Coordination Runtime (CCR) and the Decentralized Software Services (DSS). This chapter covers many of the concepts of the CCR; the next chapter discusses DSS.

To quickly clarify the differences between the CCR and DSS: The CCR is a programming model for handling multi-threading and inter-task synchronization, whereas DSS is used for building applications based on a loosely coupled service model. Services can run anywhere on the network, so DSS provides a communications infrastructure that enables services to transparently run on different nodes using all of the same CCR constructs that they would use if they were running locally.

Although you can use the CCR on its own, completely outside MRDS, this is not how you use it for creating robotics applications. Consequently, there is some overlap with DSS in this chapter because it provides an environment that makes CCR easier to use. In fact, there was some discussion during the writing of this book regarding whether CCR or DSS should be covered first. You can take a peek at the next chapter and decide for yourself in which order you want to read them.

A large amount of documentation is supplied with MRDS and the authors do not intend to reproduce it all here. You should read the online CCR User Guide (<http://msdn2.microsoft.com/en-us/library/bb905447.aspx>), which is also in the documentation that comes with MRDS. In particular, you will find the MRDS Tutorials to be an invaluable source of information.

The objective in this book is to give you a brief introduction to the important concepts and then get into coding as quickly as possible. Along the way, many new applications supplement the examples that are included in the MRDS distribution.

You will probably find that the MRDS environment is quite different from anything that you have programmed with in the past, which means that it can involve a steep learning curve. Luckily, Microsoft recognized this and built the Visual Programming Language (VPL) tool to hide many of the details about how MRDS services work. However, as a professional programmer, you need to understand what is happening “under the hood,” so the book begins with the basics, and VPL is covered later in the book.

Overview of the MRDS Framework

MRDS provides a framework for developing robotics applications. At the lowest level it is conceptually similar to the device drivers or BIOS (basic input/output system) on a PC that provide the interface to the computer hardware.

As part of MRDS, Microsoft supplies a variety of different samples that support readily available robots, but it is up to the robot manufacturers to develop and support their own code. However, MRDS provides more than just the equivalent of device drivers.

At a higher level, MRDS is similar to an operating system for robots. It is not a true operating system because MRDS services must be hosted on a Windows platform with .NET installed. In many cases this means that MRDS runs on a PC and communicates with the robot via a wireless connection. Alternatively, an embedded PC, laptop, or PDA can be mounted on the robot to run MRDS. This is an important point: You do not compile MRDS code and load it directly into a robot — there must be a Windows device somewhere, either on the robot or connected to it through a communications link.

The Need for Concurrency

Anyone who has worked with robots knows that several things are happening at the same time. For example, while your program is driving a robot around, it must also be listening to information from the sensors so that the robot does not bump into anything.

In the past, to handle this robotic multi-tasking, you would have had to write a multi-threaded application using Windows threads. This was a complex task. For example, you would have used mutual exclusion, or mutexes, to prevent two threads from simultaneously attempting to update the same variable. Semaphores and critical sections were used to control access to information about the current *state* of the robot so that the state did not become inconsistent. However, with mutexes comes the possibility of deadlocks. Debugging in a real-time environment to find race conditions that intermittently cause deadlocks was a nightmare.

The CCR eliminates many of the issues related to multi-threading. (However, you can still create deadlocks if you use the CCR inappropriately. You’ll see an example of this later in the chapter.) It uses its own threading mechanism, which is much more efficient than the Windows threading model.

Another issue with robots is that events happen asynchronously. You don’t want your service to be constantly polling the robot for sensor information. In reality, some services have to do this because the robot itself does not spontaneously send sensor updates. However, your code does not need to know about this because the details are buried inside another service that you partner with.

Similarly, you cannot afford to write timing loops that tie up the CPU. This is a common practice on cheap hobby robots because the onboard computer is quite primitive and doesn't support multi-threading. The problem with such a "busy wait" is that the robot is effectively "blind" while the CPU is executing a time delay. Provided that the delays are quite short, the robot will appear to be executing multiple operations simultaneously. On a Windows system, however, you need a mechanism to wait for a period of time that does not involve running the CPU at 100 percent in a tight loop.

Services — The Basic Building Blocks

The CCR supplies the underlying infrastructure that enables multiple tasks to execute concurrently on a single computer. DSS adds another layer for combining CCR applications, called *services*, and at the same time it enables these services to run on completely separate computers and communicate via the network.

Microsoft has defined a set of *generic contracts* that describe commonly used robotics services. These contracts specify the APIs that must be used to communicate with robot components such as motors, sonar sensors, and even webcams. By standardizing these interfaces, the robotics community can share code more easily.

For example, the generic differential drive contract enables a single program called the Dashboard (see Chapter 4) to drive around any robot that has two wheels regardless of the underlying hardware, and to do this using a mouse, a joystick, or even an Xbox controller without having to configure anything.

Services are discussed in more detail in Chapter 3, but they are so fundamental to MRDS that they need to be introduced here.

Orchestration — Putting Services Together

Every MRDS application that you build will contain one or more services. Combining these services and passing messages between them, whether they are located on the same or different computers, is one of the tasks of DSS.

Figure 2-1 shows an example of how the services might be *orchestrated* to control a robot. Combining services, a process called *partnering*, is one of the topics in the next chapter. It is the job of the orchestration service to implement high-level behaviors, such as following a line or solving a maze.

You might wonder why DSS is not shown sitting on top of the CCR in the Runtime Environment in Figure 2-1. The reason is very simple: Many services make direct calls to the CCR. Clearly, the CCR is too low level to provide orchestration; it just provides the machinery to make it all happen.

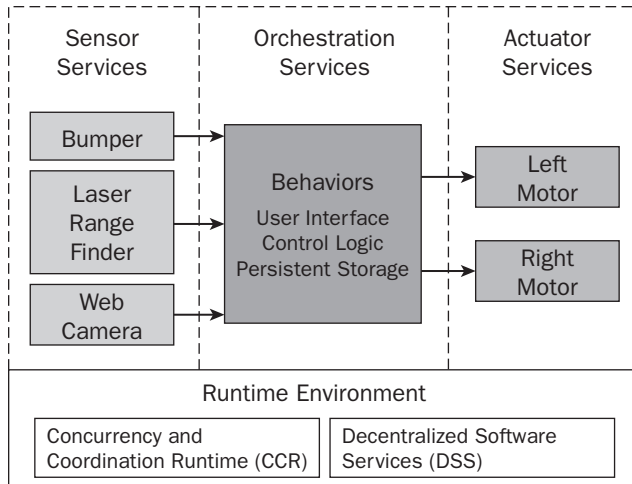


Figure 2-1

Setting Up for This Chapter

Although MRDS is based on the CCR, it is possible to use the CCR on its own. However, you need the combination of the CCR and DSS in order to write MRDS services. Therefore, the examples in this chapter are in the context of a DSS service. (DSS services are covered in Chapter 3.)

You have two options at this stage:

- ❑ You can simply open the `CCRExamples` solution in the `ProMRDS\Chapter2` folder.
- ❑ If you learn best by doing things and you like typing, then you can enter the code in this chapter into an empty service and then build and run it.

If you have successfully installed the software for this book, then you already have a folder called `ProMRDS` under your MRDS installation (by default it is in `C:\Microsoft Robotics Studio (1.5)`). If you cannot find this folder, then please go back to Chapter 1 and follow the installation instructions.

Throughout this book, you will continually see references to the `ProMRDS` folder. Remember that this is under the MRDS installation point, or what is called the *root* or the *mountpoint* in MRDS terminology.

We don't really believe that people like typing in code from a textbook, but even if you do, you might want to check out the sample code first, as explained below. It can also act as a reference in case you can't get your code to work due to some small typing error.

An Important Note about Versions

At the time of writing, Microsoft had just released Visual Studio 2008. However, all of the example code for this book was developed using Visual Studio 2005. Both of these versions can coexist on the same computer, but you should use VS 2005 if possible.

In addition, the code in this book is based on MRDS Version 1.5 with the December 2007 Refresh applied. Planning was already underway for Version 2.0 as this book was being written; a new Community Technology Preview (CTP) of 2.0 will be available by the time this book is published. At the same time, Microsoft plans to rename the product to Microsoft Robotics Developer Studio — an author's nightmare in the final stages of writing a book! Therefore, you see references to MRDS in this book, instead of MSRS. Consider them to be the same product.

You should download and install MRDS V1.5 even if a later CTP version of MRDS is available. Multiple versions can be installed on the same computer and can operate side-by-side.

Visit the website for this book for updates at www.proMRDS.com. VS 2008 versions of the code will be posted to this site as well as updates when MRDS V2.0 is officially released. You can also get downloads and errata at www.wrox.com.

Using the CCRExamples Project

Navigate to the `ProMRDS\Chapter2\CCRExamples` folder using Windows Explorer. Double-click on `CCRExamples.sln` to open the solution in Visual Studio.

If you are using the Express Edition of Visual C# and you have other Express products installed, then you might have to select C# from the pop-up dialog. The Express Edition does not properly recognize the solution file.

Once you have the solution open in Visual Studio, open the file `CCRExamples.cs` and locate the method called `Start`. All DSS services have a method with this name and it is called during the initialization of the service.

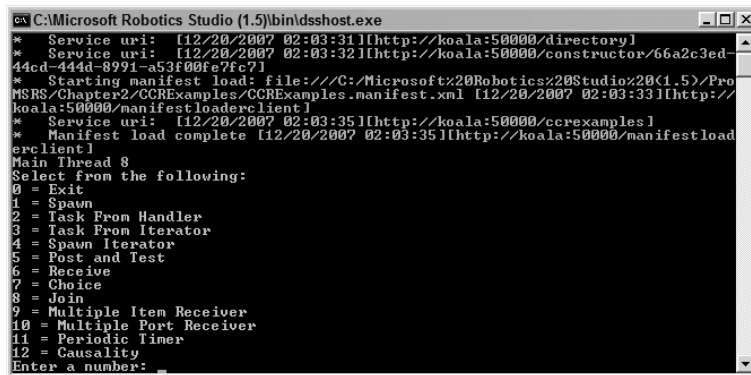
Note that the code uses a lot of `Console.WriteLine` statements to display information in the MRDS DOS command window that is created automatically when you run the project. Using `Console.WriteLine` is not recommended because MRDS has other ways of displaying output. However, for these examples it is easier to just use the console.

The code in the `Start` method runs in a loop that displays a menu and asks you to enter a number for one of the examples. Then it executes the selected example code. If you enter a value of zero, then the service exits. There is no need to discuss this part of the code because it has nothing to do with the CCR or DSS.

Chapter 2: Concurrency and Coordination Runtime (CCR)

To try out the examples, select Start Without Debugging from the Debug menu or press Ctrl+F5. You need to run without the debugger for the Causality example to work correctly, as explained below. All the other examples can be run with the debugger.

The screenshot in Figure 2-2 shows the initial output from the service. Note that the messages at the top of the window are from DSS as it starts the service. The first message from the program is “Main Thread 8,” which indicates that it is running. The text menu at the bottom of the screen provides a list of the available examples. These cover the major features of the CCR and are explained in the rest of this chapter.



```
C:\Microsoft Robotics Studio (1.5)\bin\dsshost.exe
* Service uri: [12/20/2007 02:03:31][http://koala:50000/directory]
* Service uri: [12/20/2007 02:03:32][http://koala:50000/constructor/66a2c3ed-44cd-444d-8991-a53f00fe7fc7]
* Starting manifest load: file:///C:/Microsoft%20Robotics%20Studio%20(1.5)/Pro
MRS/Chapter2/CCRExamples/CCRExamples.manifest.xml [12/20/2007 02:03:33][http://
koala:50000/manifestloaderclient]
* Service uri: [12/20/2007 02:03:35][http://koala:50000/ccrexamples]
* Manifest load complete [12/20/2007 02:03:35][http://koala:50000/manifestload
erclient]
Main Thread 8
Select from the following:
0 = Exit
1 = Spawn
2 = Task From Handler
3 = Task From Iterator
4 = Spawn Iterator
5 = Post and Test
6 = Receive
7 = Choice
8 = Join
9 = Multiple Item Receiver
10 = Multiple Port Receiver
11 = Periodic Timer
12 = Causality
Enter a number: _
```

Figure 2-2

Entering the Code Manually

If you want to enter the code yourself, then start by opening Visual Studio and creating a new DSS service. MRDS installs templates so that you can easily create new projects.

1. Start Visual Studio 2005 and then select File ⇨ New ⇨ Project.

The New Project dialog appears, as shown in Figure 2-3. Notice that the Robotics templates are selected from the Project Types under Visual C# and that Simple Dss Service (1.5) is highlighted.

If you don't see the Robotics folder under Project Types, then it is likely that you installed Visual Studio after MRDS. In that case, you should reinstall MRDS.

You can create the service wherever you like, but it is a good idea to keep your code separate from the MRDS distribution. In Figure 2-3, the location is set to the Projects folder under the MRDS mountpoint.

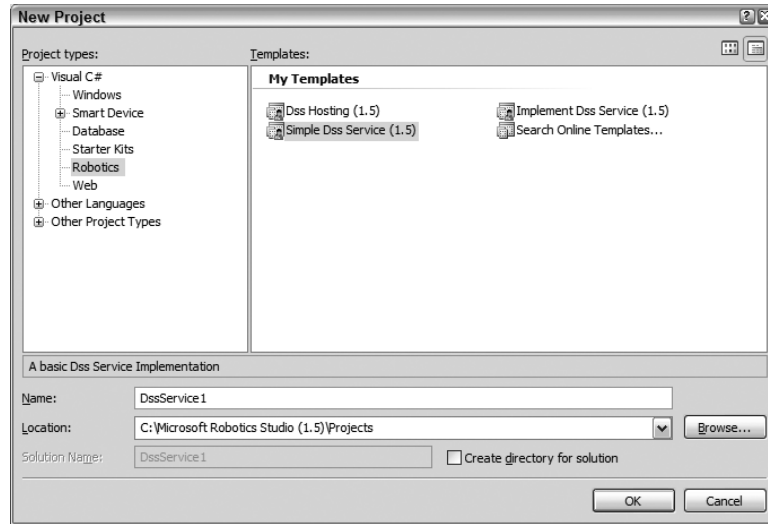


Figure 2-3

2. Give the project a name. It doesn't matter what you call it, but leave it as DssService1 for now.

If you are using Visual C# Express Edition, then the dialog is slightly different and you are not asked for the location. The project is automatically created in the Projects folder under MRDS. The dialog for the Express Edition is shown in Figure 2-4.

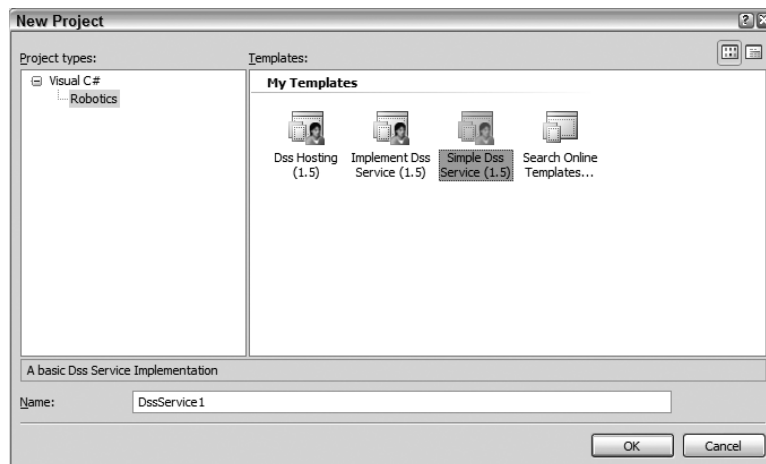


Figure 2-4

Chapter 2: Concurrency and Coordination Runtime (CCR)

3. Examine the contents of the solution in the Solution Explorer. Once the project has been created, you should see the files shown in Figure 2-5 in the Solution Explorer. Note that the References are expanded to show that `Ccr.Core`, `DssBase` and `DssRuntime` were added automatically. These references must always be included for MRDS service projects.

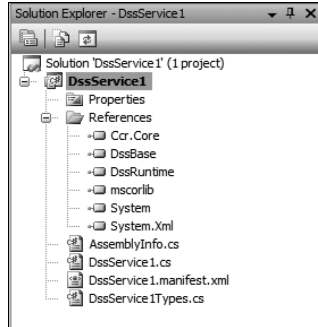


Figure 2-5

4. Add a new using statement. To do so, open the main service source file, which is `DssService1.cs`. At the top of the file you will find several using statements. Add one for `System.Threading` as shown here:

```
using ...
using dssservice1 = Robotics.DssService1;
using System.Threading;
```

5. Scroll through the code to update the `Start` method. It should look like the following:

```
/// <summary>
/// Service Start
/// </summary>
protected override void Start()
{
    base.Start();
    // Add service specific initialization here.
}
```

`Start` is called when the service is started by DSS. This is where you would normally place code to perform any initialization that your service requires, but you are going to use it to execute a series of examples.

Note that in the examples that follow, each example is shown as a separate procedure. You can add each procedure to the service one at a time and put a call to it in the `Start` method. This helps to keep the code compartmentalized and easier to manage. Eventually, you will end up with a service that runs all of the examples. (The `CCRExamples` code also has regions around each of the examples as a further way of managing the code.)

To reiterate, you need to enter the code for each of the example functions after the end of the `Start` method and then insert an appropriate call to the example function just below `base.Start` where the comment tells you to enter your own initialization code, like so:

```
protected override void Start()
{
    base.Start();
    // Add service specific initialization here.

    // Wait until the service has started up completely.
    Wait(2000);
    Console.WriteLine("Main Thread {0}", Thread.CurrentThread.ManagedThreadId);

    // Call the Example
    ExampleFunction();
}

void ExampleFunction()
{
    // Example code goes here
}
```

After you insert the preceding code, compile the solution and run it in the debugger to ensure that everything is OK. Of course, it is much easier if you just open the CCRExamples solution.

Tips for Coding with MRDS

Most programs are not written from scratch. Programmers often find a similar piece of code and then copy, paste, and modify it as required. The same is true of MRDS. There are literally dozens of examples provided in the MRDS `samples` folder, and of course there are the examples from the website for this book (which you installed into the `PromRDS` folder). Novel examples are popping up on the Web all the time, such as driving your robot using a Nintendo Wii remote controller with the .NET managed library for the Wiimote as it is called (available from www.codeplex.com/WiimoteLib). Do a Google search for “MSRS wiimote” for more information.

The following are some tips for coding with MRDS:

- ❑ You should definitely read the documentation available online at <http://msdn2.microsoft.com/en-us/library/bb881626.aspx>. This documentation is also supplied as a compiled help file, called `MSRSUserGuide.chm`, in the MRDS documentation directory. (The online version might be more current.)
- ❑ If you thought that that .NET Framework Class Library was huge, now you have to add to that the CCR and DSS libraries. You should make extensive use of IntelliSense and the Object Browser in Visual Studio to find out what properties and methods are available for the various classes. You can also right-click a class name in the code and select `Go To Definition` to see the properties and methods of a class via reflection.
- ❑ No attempt is made in this book to cover all of the available classes because it would be a waste of paper and you would not want to read it anyway. All the class information is at your fingertips. An online class reference for MRDS is accessible from Visual Studio by pressing F1 while you have the cursor on a CCR or DSS class in the editor. It is also available through the online documentation page. The class reference was released while this book was being written and it was still very basic, with very few examples.

Coordination and Concurrency Runtime

As previously mentioned, the CCR is a lightweight library that is supplied as a .NET DLL. It is designed to handle asynchronous communication between loosely coupled services that are running concurrently. This chapter contains many concepts that may be new to you, so you might want to skim through it on the first reading and then come back later to reread portions of it more carefully. It is difficult to cover all of the material sequentially because various concepts are interrelated. Figure 2-6 summarizes the CCR architecture. The following list briefly outlines how the CCR works. Then it is described in more detail throughout the rest of the chapter. You can refer back to Figure 2-6 as you read through the rest of the sections.

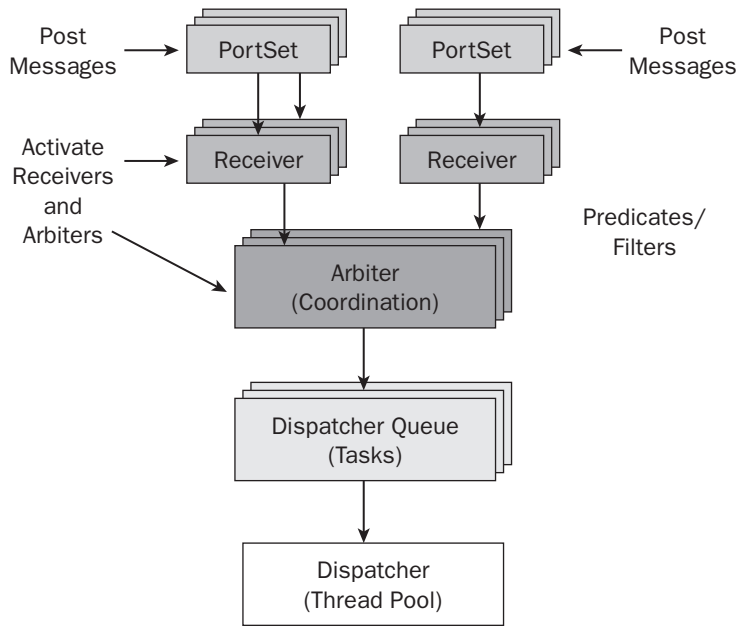


Figure 2-6

- ❑ The CCR uses an asynchronous programming model based on *message passing* using a structure called a *port*. A *message* is an instance of any data type that the CLR can handle. When you write DSS services, you declare your own classes for message types. If messages have to be sent across the network, then they are first serialized into XML and then deserialized on the destination node.
- ❑ Ports are message queues that only accept messages of the same data type as the one in the port declaration. This “type safety” is important for ensuring that obvious mistakes are picked up at compile time.
- ❑ In CCR terminology, you place messages into a port by *posting* to the port. Messages remain in ports until they are dequeued by a *receiver*. Activation conditions can be set on receivers to create complex logical expressions, such as a *join* between two ports (two messages must arrive, which is effectively a logical AND) or a *choice* between two ports (a message can arrive on either port, creating a logical OR). Evaluating activation conditions is the job of *arbiters*.

- ❑ The coordination primitives, implemented through arbiters, are used to synchronize and control *tasks*. Once a receiver's conditions have been met, a task is queued to a *dispatcher queue* and then passed to a *dispatcher* for execution.
- ❑ When a task is scheduled to run, code called a *handler* is executed. Handlers are pieces of code that run in a fully multi-threaded environment. They can run on their own if you simply request the execution of a task, or more commonly they consume messages as a consequence of being connected to a receiver. Handlers often generate new messages, and the cycle continues.
- ❑ There are mechanisms within the CCR to handle failures, also called *faults*, in relation to web requests or *exceptions* in the usual .NET CLR sense. A structured approach called *causalities* is analogous to try/catch.

Most of the time, the CCR handles all this transparently, and there is no need for you to explicitly define mutexes or write callback procedures, which you might be used to using for multi-threaded programming in the past. The standard Windows synchronization primitives still work, but they are not usually needed.

The DSS is based on the CCR and does not require any other components. It provides a hosting environment for running services and making them accessible via a web browser. You cannot separate DSS from the CCR. When you run the examples in this chapter, you are running a DSS node and executing a service inside this node.

Concurrent Execution

Obviously, a key feature of the CCR is that it supports multi-threading, or *concurrency*. Understanding how concurrency works in the CCR environment is your first challenge as you learn about MRDS.

However, before examining the components of Figure 2-6, you need to understand a few key concepts: tasks, delegates and iterators. However, in order to do this, you need to use some CCR APIs that have not been discussed yet — a classic “chicken or the egg” situation.

Tasks

At the bottom of Figure 2-6, you can see that dispatchers schedule tasks by allocating them to threads from a pool. A task contains a reference to a handler that is a piece of code you want executed.

In the CCR, tasks implement the `ITask` interface. This is useful because it gives tasks a data structure, enabling them to be passed around and queued. However, it also means that you must wrap a piece of code as an `ITask` before you can submit it to a dispatcher queue.

Running a task is relatively easy. Consider the following simple handler that loops around counting from 1 to 10, with a delay in between printing the numbers:

```
// Simple Handlers do not take parameters and do not return anything
void SimpleHandler()
{
    // Handler does not really do anything
    Console.WriteLine("Simple Handler Thread {0}",
        Thread.CurrentThread.ManagedThreadId);
}
```

(continued)

Chapter 2: Concurrency and Coordination Runtime (CCR)

(continued)

```
    for (int i = 0; i < 10; i++)
    {
        Wait(100);
        Console.Write(i + " ");
    }

    // Try adding PressEnter() and see what happens.
    // It gets complicated!
    //PressEnter();

    Console.WriteLine("Finished Simple Handler Thread {0}",
        Thread.CurrentThread.ManagedThreadId);
}
```

This is not very exciting on its own, but things get more interesting when you run this code several times concurrently.

The `DsspServiceBase` class (part of DSS) provides a wrapper to execute a handler as a task called `Spawn`. If you select the `Spawn` option from the menu in `CCRExamples`, it executes the following code:

```
Spawn(SimpleHandler);
Spawn(SimpleHandler);
Spawn(SimpleHandler);
Spawn(SimpleHandler);
Console.WriteLine("Spawns executed ...");
```

If you are creating your own service, then place this code into the `Start` method and add the `SimpleHandler` method to your service.

The output should look something like the following:

```
Simple Handler Thread 8
Spawns executed ...
0 1 2 3 Select from the following:
0 = Exit
1 = Spawn
2 = Task From Handler
3 = Task From Iterator
4 = SpawnIterator
5 = Post and Test
6 = Receive
7 = Choice
8 = Join
9 = MultipleItemReceive
10 = MultiplePortReceive
11 = Periodic Timer
12 = Causality
Enter a number: 4 5 6 7 8 9 Finished Simple Handler Thread 8
Simple Handler Thread 8
0 1 2 3 4 5 6 7 8 9 Finished Simple Handler Thread 8
Simple Handler Thread 8
```

Chapter 2: Concurrency and Coordination Runtime (CCR)

```
0 1 2 3 4 5 6 7 8 9 Finished Simple Handler Thread 8
Simple Handler Thread 8
0 1 2 3 4 5 6 7 8 9 Finished Simple Handler Thread 8
```

The `SimpleHandler` displays its first message before all of the spawns have completed. Then it starts to count from 1 to 10, but the main thread displays the menu again so the two are interspersed in the console output. Eventually the first task completes, and then the handler runs three more times. Meanwhile, the main thread is quietly waiting in the background for you to select another menu option, effectively tying up a thread.

Notice in this case that all of the spawned threads are on thread number 8. How can this be in a multi-threaded environment? The answer is very simple: The default number of threads in the pool in this case is only two. The first thread executes the `Spawn` statements and displays the menu, so it is tied up. That leaves only one thread to execute the spawned tasks, which therefore have to execute sequentially.

This behavior is not something that you can count on. If the computer had a four-core processor, then DSS would have automatically created a dispatcher with four threads and all of the tasks would have executed simultaneously. Try the following example to prove this.

Assuming that you saw the same thread used repeatedly, you can change the attributes on the service so that more threads are created in the initial pool. (If you did *not* see this behavior, then you probably have a four-core processor, or two dual-core processors, etc.) This is really a DSS issue because you are changing attributes on a `DsspServiceBase` class, but it makes sense to discuss it here.

At the top of the code, locate the service class definition and add the `ActivationSettings` attribute. Set the parameters to `false` for `ShareDispatcher` and `6` for `ExecutionUnitsPerDispatcher` as shown:

```
/// <summary>
/// Implementation class for CCRExamples
/// </summary>
[DisplayName("CCRExamples")]
[Description("The CCRExamples Service")]
// Add the following attribute to the service to create six threads
// instead of the default, which is probably only two
[ActivationSettings(ShareDispatcher=false, ExecutionUnitsPerDispatcher=6)]
[Contract(Contract.Identifier)]
public class CCRExamplesService : DsspServiceBase
{
    ...
}
```

Recompile and run the service again. Select the `Spawn` example and see what happens. The output should be as follows:

```
Spawns executed ...
Simple Handler Thread 11
Simple Handler Thread 13
Simple Handler Thread 14
Simple Handler Thread 15
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3 Select from the following:
0 = Exit
```

(continued)

Chapter 2: Concurrency and Coordination Runtime (CCR)

(continued)

```
1 = Spawn
2 = Task From Handler
3 = Task From Iterator
4 = SpawnIterator
5 = Post and Test
6 = Receive
7 = Choice
8 = Join
9 = MultipleItemReceive
10 = MultiplePortReceive
11 = Periodic Timer
12 = Causality
Enter a number: 4 4 4 4 5 5 5 5 6 6 6 6 7 7 7 7 8 8 8 8 9 Finished Simple ◀
Handler Thread 14
9 Finished Simple Handler Thread 13
9 Finished Simple Handler Thread 11
9 Finished Simple Handler Thread 15
```

All of the spawns execute simultaneously, and the menu still runs too! You can see this because each number appears four times in a row.

This is an important point: The default DSS dispatcher thread pool only has two threads for a single CPU machine (even if it is dual-core). The minimum number of threads is two.

Before you continue, go back and comment out the `service` attribute. If you don't do this, then some of the remaining examples will have different output from what is shown in the book.

Another way to get overlapped behavior is to create your own dispatcher so you can specify the number of threads explicitly. Add a `RunFromHandler` function as shown in the following example. It creates a new dispatcher with three threads in the pool. It then starts four tasks using the same `SimpleHandler` as above.

At this stage you don't need to know much about the dispatcher and dispatcher queue, but notice the use of `Arbiter.Activate` to activate the tasks and `Arbiter.FromHandler` to create the necessary task instances.

In the CCR you need to *activate* a task to get it to execute, which means enqueueing it to a dispatcher queue. Without `Arbiter.Activate`, the `Arbiter.FromHandler` would do nothing — it would simply create an instance of an `ITask`, but not schedule it for execution. This is a potential trap for novices.

```
// Run a Task from a Handler
void RunFromHandler()
{
    // Explicitly create a Dispatcher so we can control the pool size
    Dispatcher d = new Dispatcher(3, "Test Pool");
    DispatcherQueue q = new DispatcherQueue("Test Queue", d);

    // Activate FOUR tasks with a pool of THREE threads
```

```
Arbiter.Activate(q,
    Arbiter.FromHandler(SimpleHandler),
    Arbiter.FromHandler(SimpleHandler),
    Arbiter.FromHandler(SimpleHandler),
    Arbiter.FromHandler(SimpleHandler));
}
```

When you run this code (the menu option in CCRExamples is Task From Handler), the output is quite different from the example with `Spawn`:

```
Simple Handler Thread 18
Simple Handler Thread 19
Simple Handler Thread 17
0 0 0 1 1 1 2 2 2 3 3 3 Select from the following:
0 = Exit
1 = Spawn
2 = Task From Handler
3 = Task From Iterator
4 = SpawnIterator
5 = Post and Test
6 = Receive
7 = Choice
8 = Join
9 = MultipleItemReceive
10 = MultiplePortReceive
11 = Periodic Timer
12 = Causality
Enter a number: 4 4 4 5 5 5 6 6 6 7 7 7 8 8 8 9 Finished Simple Handler Thread 19
Simple Handler Thread 19
9 Finished Simple Handler Thread 17
9 Finished Simple Handler Thread 18
0 1 2 3 4 5 6 7 8 9 Finished Simple Handler Thread 19
```

Notice this time that three different threads start up straight away and announce themselves. They begin counting together, with groups of three numbers appearing on the screen. Then the menu is redisplayed by the main thread, which is not part of the new dispatcher's thread pool. Eventually thread 19 finishes, although it was not the first to start, and it starts executing the fourth task while threads 17 and 18 finish off.

This whole process is nondeterministic. Try running it several times and see which thread starts first, and which one finishes first. You cannot predict it. All you can say is that eventually all four tasks will be completed.

If you need even more proof that the CCR is multi-tasking, try the periodic timer example later in this chapter. You can actually select another menu option while the timer is running and you will see that "Tick" messages are injected into the output from the other example.

Delegates

Quite often a task is just a few lines of code. The simplest way to implement this code fragment is using a C# *delegate* to construct an anonymous method. A delegate is similar to a normal procedure declaration except that the procedure name is the keyword `delegate`. (Delegates do not exist in other .NET languages and you have to define a normal procedure instead.)

Chapter 2: Concurrency and Coordination Runtime (CCR)

If you have not seen delegates before, here is a code fragment that illustrates the use of delegates. In this example, a message is sent to move a robotic arm and the `Choice` arbiter waits for either a response message to say that the move was successful or a `Fault` message indicating that there was some problem. (Don't worry about the rest of the syntax; `Choice` is discussed later in the chapter.)

```
_mainPort.Post(moveCommand);

bool success = true;

yield return Arbiter.Choice(
    moveCommand.ResponsePort,
    delegate(SSC32ResponseType response)
    {
    },
    delegate(Fault f)
    {
        LogError(f);
        LogError("Servos cannot be initialized.");
        success = false;
    }
);
```

In the preceding example, the first delegate has no code — it is just a placeholder. You don't need it for any purpose other than to be there when the move is successful. However, in general, there will be some code in this delegate. The second delegate, however, logs the error and sets a flag to say that the move failed.

Look carefully at the code. Notice that the `success` variable is defined outside of the delegates. However, the second delegate can still access it and update its value. This is a powerful feature of delegates when used in this way, but one that is only available in C#. (Other languages have to use global variables to achieve the same effect.)

Technically, this is what is called a lexical closure. If you have programmed in an asynchronous environment before, then an I/O completion routine is another form of closure. It has been common in Lisp-based languages for some time and there is a similar construct in Ruby. However, it is relatively new in imperative languages. Interestingly, this concept is on the drawing board for the next version of Java.

Think about it for a moment. The `Choice` arbiter has executed a delegate, probably on a different thread, and it has updated the local variable called `success` in the current procedure. That's cool stuff, but potentially dangerous.

Usually, a delegate contains only a small amount of code, but it can call other procedures and be quite complex. A key point in multi-threaded design, though, is to try to keep your tasks short and ensure that they do not block.

Another concept illustrated in the preceding code that might be new to you is `yield return`. This is called a *continuation*. The code is suspended at this point, and some time later it resumes execution with the next statement after the `yield return`. The next section discusses this further.

Iterators

A key concept for the CCR is the use of an *iterator*, which allows sequential execution of code but without blocking the execution thread when it needs to wait for a message. When an operation must be performed that will take some unknown amount of time to execute, the iterator effectively remembers the current location in the code and then relinquishes control until a message is received. At some later point in time, the message arrives and the code resumes execution from the point where it left off.

Iterators were introduced to C# in version 2.0 to support `foreach` iteration without having to implement the entire `IEnumerable` interface. Each time that an iterator is called, it is supposed to return the next value in the sequence using `yield return`, or terminate the iteration using `yield break`. Other .NET languages do not have this support (yet) and so it is currently much easier to program with the CCR in C#.

The CCR uses iterators based on a sequence of tasks to enable you to write code that looks like it executes sequentially, but in reality it is a series of asynchronous steps. An iterator is declared as a method of type `IEnumerator<ITask>`, which means it will iterate over tasks. The syntax `<ITask>` is known as a generic and is discussed further next. The compiler also recognizes that this type of method can contain `yield return` and `yield break` statements, which are not valid in normal code.

Generics

C# uses generic classes, which are denoted by the syntax `<type>`. These are similar to templates in C++ if you are familiar with them.

In the case of `IEnumerator<ITask>`, the function is declared to be an enumerator that returns objects of type `ITask`. Other enumerators can also be built that return different types, but you only need to use tasks for MRDS. Also notice the syntax for declaring a `Port` that receives integer values as messages:

```
Port<int>
```

You can use any .NET CLR data type, including user-defined classes, as the message type for a `Port`.

Consider the following routine, which waits on messages using two different ports. The waiting is done using `yield return`:

```
private static IEnumerator<ITask> IteratorExample()
{
    Port<int> p1 = new Port<int>();
    Port<int> p2 = new Port<int>();

    p1.Post(0);

    bool done = false;

    while (!done)
```

(continued)

(continued)

```
{
    yield return Arbiter.Receive(false, p1,
        delegate(int i)
        {
            Console.WriteLine("P1 Thread {0}: {1}",
                Thread.CurrentThread.ManagedThreadId, i);
            p2.Post(i + 1);
        }
    );

    yield return Arbiter.Receive(false, p2,
        delegate(int i)
        {
            Console.WriteLine("P2 Thread {0}: {1}",
                Thread.CurrentThread.ManagedThreadId, i);
            if (i >= 10)
                done = true;
            else
                p1.Post(i + 1);
        }
    );
}

yield break;
}
```

This looks like normal sequential code, but each time `yield return` is executed it returns a `Task` instance and remembers the location it reached in the code so that execution can resume from the following statement the next time the iterator is called (hence the name “continuation”). The CCR keeps calling the iterator, executing the sequence of tasks it returns, until eventually `yield break` is executed and the iteration stops.

Notice that the last line of the routine is `yield break`. Because this is the end of the routine, there is really no reason to add this statement. However, sometimes the compiler cannot determine whether you have finished the code or not, and it complains with an error that “Not all code paths return a value.”

In this case, the returned tasks are *receivers*. (Receivers are discussed in the section “Receivers and Arbiters” later in the chapter.) For now, all you need to know is that a receiver waits for a message to arrive on a port. While the receiver is waiting, the operating system thread is freed to do other things (or the CPU will be idle if no tasks are waiting to execute).

Some Key Points about Iterators

One of the major concerns in multi-threaded programming is the blocking of threads. This is highly undesirable because it takes the thread out of circulation for the period that it is blocked. In the `IteratorExample` code, it appears that the `Arbiter.Receive` blocks execution, but in fact the thread is not blocked at all: It is returned to the thread pool until the receiver executes.

Second, and this is again jumping ahead a little here, the receivers in the example are nonpersistent. (The first parameter in the `Arbiter.Receive` is `false`.) This is essential for an iterator to work because the `yield` return must wait for completion. If you specify a CCR arbiter that is persistent, its work is never finished because it remains available to process new messages. In that case, the `yield` return will never resume execution of the procedure.

Under the covers, the C# compiler rewrites an iterator into an invisible class with several methods. One of these methods is `MoveNext`, which is part of the machinery that handles advancing to the next code segment after a `yield` return. You don't need to know about this to use an iterator, but it is interesting.

Note that iterators have to be executed as tasks. If you simply insert a call to the `IteratorExample` method in your code, nothing will happen. There are no compilation errors, but the code is not executed.

The `yield` statement cannot be used inside a delegate; inside a `catch` block; or in a `try` if it has an associated `catch`. In fact, it cannot be used anywhere except inside an iterator. It causes compilation errors when used in normal code.

Iterators cannot return values in `out` or `ref` parameters so you have to use global variables if you want to return values to the caller.

In `CCRExamples`, the preceding `IteratorExample` code can be executed by selecting the Task From Iterator example, which executes the following code:

```
void RunFromIterator()
{
    Dispatcher d = new Dispatcher(4, "Test Pool");
    DispatcherQueue taskQ = new DispatcherQueue("Test Queue", d);

    Console.WriteLine("Before Iterator submitted - thread {0}",
        Thread.CurrentThread.ManagedThreadId);

    Arbiter.Activate(taskQ,
        Arbiter.FromIteratorHandler(IteratorExample),
        Arbiter.FromIteratorHandler(IteratorExample));

    Console.WriteLine("After Iterator submitted - thread {0}",
        Thread.CurrentThread.ManagedThreadId);
}
```

This code creates a new dispatcher with four threads and then activates the iterator twice by constructing a task array using `Arbiter.FromIteratorHandler`.

Chapter 2: Concurrency and Coordination Runtime (CCR)

The output from the example should look something like the following, but you will find that the order of the output changes every time you run it:

```
Before Iterator submitted - thread 9
After Iterator submitted - thread 9
P1 Thread 20: 0
P2 Thread 22: 1
P1 Thread 21: 0
P2 Thread 21: 1
P1 Thread 21: 2
P2 Thread 21: 3
P1 Thread 21: 4
P2 Thread 21: 5
P1 Thread 21: 6
P2 Thread 21: 7
P1 Thread 20: 8
P1 Thread 23: 2
P2 Thread 23: 3
P1 Thread 23: 4
P2 Thread 23: 5
P1 Thread 23: 6
P2 Thread 23: 7
P1 Thread 23: 8
P2 Thread 23: 9
P1 Thread 23: 10
P2 Thread 21: 9
P1 Thread 21: 10
P2 Thread 21: 11
P2 Thread 23: 11
```

In the output, you can see that the iterators are submitted on thread 9. However, threads 20 to 23 are used to do the work of the iterators because a new dispatcher is specified in the `Arbiter.Activate`. The iterators post messages backward and forward between their two ports, `p1` and `p2`, causing the execution to bounce between two threads.

The threads run as fast as their little feet will carry them (threads have much smaller feet than Hobbits). However, they do not all run at the same speed. They start out counting in synch, 0, 1, 0, 1, but then thread 21 gets a burst of speed, almost finishes the iterations, and unfortunately it hands the baton to thread 20, who fumbles it. Thread 23 jumps in and tries to finish the race, but is eventually beaten by thread 21. That's about as exciting as the CCR gets.

Similar to what you saw earlier with `Spawn`, there is also a `SpawnIterator` method provided by DSS. An example is included in `CCRExamples` as follows:

```
// A slightly different iterator with a parameter for SpawnIterator
private static IEnumerable<ITask> SpawnIteratorExample(int n)
{
    Port<int> p1 = new Port<int>();
    Port<int> p2 = new Port<int>();

    p1.Post(n);

    bool done = false;
```

Chapter 2: Concurrency and Coordination Runtime (CCR)

```
while (!done)
{
    yield return Arbiter.Receive(false, p1, delegate(int i)
    {
        Console.WriteLine("P1 Thread {0}: {1}", ←
Thread.CurrentThread.ManagedThreadId, i);
        p2.Post(i + 1);
    });

    yield return Arbiter.Receive(false, p2, delegate(int i)
    {
        Console.WriteLine("P2 Thread {0}: {1}", ←
Thread.CurrentThread.ManagedThreadId, i);
        if (i >= n+10)
            done = true;
        else
            p1.Post(i + 1);
    });
}

yield break;
}
```

This code differs slightly from the previous code because the iterator accepts a parameter, which is the number to start counting from.

The corresponding code to launch the iterator is as follows:

```
SpawnIterator<int>(5, SpawnIteratorExample);
Console.WriteLine("Spawn Iterator executed ...");
```

Notice that the call to `SpawnIterator` has two parameters, and that the type of the first parameter is declared using a generic `<int>`. `Spawn` and `SpawnIterator` each have three overloads that accept one, two, or three parameters, which are passed to the handler. There are several similar overloads for common `Arbiter` APIs so that you can pass parameters to handlers.

When the code is executed (select the `SpawnIterator` example in `CCRExamples`), the output should be as follows. Note that again a single thread is used in all cases because `SpawnIterator` uses the default DSS dispatcher and there are only two threads in the pool:

```
Spawn Iterator executed ...
P1 Thread 8: 5
P2 Thread 8: 6
P1 Thread 8: 7
P2 Thread 8: 8
P1 Thread 8: 9
P2 Thread 8: 10
P1 Thread 8: 11
P2 Thread 8: 12
```

(continued)

(continued)

```
P1 Thread 8: 13
P2 Thread 8: 14
P1 Thread 8: 15
P2 Thread 8: 16
```

You need to become familiar with iterators because they are used extensively in the MRDS sample code.

Ports and Messages

The most important class in the CCR is the `Port`, which is basically a First-In-First-Out (FIFO) queue for *messages*.

Messages, also referred to as *requests* or *responses*, are just objects of a specified type. You can create your own classes and send instances of these classes as messages. (In fact, this is how services work, and their APIs are defined in terms of the *operations* they can perform, which are implemented using different classes of messages. There is more on this in the next chapter.)

Consider the following simple port, for example:

```
Port<int> intPort = new Port<int>();
intPort.Post(42);
```

In the preceding code snippet, the `Port` class creates a new port that accepts only integers. Because `Port` is a generic class, you can just as easily create a port for strings, doubles, or even more complex objects such as lists or classes that you define yourself.

This code creates a new port called `intPort`, and sends the value `42` to it as a message using the `Post` method. Because `intPort` has an explicit type of `int`, you can only *post* (enqueue) integer values to it. If you attempt to post a `string`, then you will get a compilation error. This type-checking helps to eliminate subtle bugs.

The posted value of `42` remains in the port queue until it is dequeued either by being explicitly read or by a *receiver*. (Receivers are covered in the next section.)

If messages are never removed from the port, then they just keep accumulating, which poses a potential memory leak. You can empty a port by calling its `Clear` method. It is also possible to set policies that prevent the build-up of messages in a port (as explained later).

Caution on Modifying Messages

You might assume that once you post a message, you can forget about it and dispose of the original variable or reuse it. That would be a bad mistake! The memory for the message is still in use in the port queue until the message has been processed.

When posting messages in a loop, you have to create a new instance of the message each time around the loop. If you are concerned about memory usage, then you can dispose of the message in the handler once you have finished with it. Otherwise, the CLR

garbage collector will eventually determine that the message has no remaining references and reclaim the memory.

Another common mistake is modifying the contents of a message object (usually a class instance) before it has been received. This affects the message that has already been sent. The simple solution is to always create new messages.

The major advantage of ports is that if you have a handle to a port — for example, if it is a global variable — then you can post messages to it from any thread and it will always be a safe operation. Furthermore, if all the receivers are busy, the message simply waits until it can be processed. The sender of the message does not have to wait because posting does not block (except for certain policies).

Reading from a Port

To see what the port contains (for debugging purposes), you can use the `ToString` method:

```
// Display the state of the port
Console.WriteLine(intPort.ToString());
```

A port has an `ItemCount` property that can be used to determine how many items are queued. However, if you want to see whether an item is available and retrieve it (if there is one), you can use the `Test` method, which returns the item as an object, or `null` if there is no item available. An overloaded version returns a Boolean result and passes back the item (or `null` if there is no item) as an `out` parameter. Note that `Test` removes the item atomically, i.e., in a thread-safe manner, so that multiple threads cannot inadvertently remove the same item.

```
int j;
// Use Test to read from the port (or fail)
if (intPort.Test(out j))
    Console.WriteLine("Got value: " + j);
else
    Console.WriteLine("No value!");
```

This is a straightforward way to use a port as a FIFO queue that can be fed data from anywhere in the application. However, your code would have to continually poll the port to determine whether new items were available — for example, by using timer events. This is not the way that ports are usually used; it is more common to set up receivers that execute tasks automatically when messages arrive.

If you want to flush a port, you can execute a loop that keeps removing items using `Test` until no more are available. However, it is much more efficient just to use `Clear`.

Here is a complete example that uses a port and the `Test` method. You should be able to follow it based on the discussion so far. Enter the code into your service just below the `Start` method if you are building the code yourself, and then put a call to `PostAndTest` inside the `Start` method. If you are using the `CCRExamples` service, you can run the service and select the `Post` and `Test` item from the menu (refer to Figure 2-2).

Chapter 2: Concurrency and Coordination Runtime (CCR)

```
// Post messages and use Test to retrieve them
void PostAndTest()
{
    int i, j;
    // Create a new integer port
    Port<int> intPort = new Port<int>();
    // Post the answer to life the universe and everything
    intPort.Post(42);
    Console.WriteLine("Posted a value");

    // Display the state of the port
    Console.WriteLine(intPort.ToString());

    // Use Test to read from the port (or fail)
    if (intPort.Test(out j))
        Console.WriteLine("Got value: " + j);
    else
        Console.WriteLine("No value!");

    // View the port status again
    Console.WriteLine(intPort.ToString());
    // Try Test a second time with nothing to read
    if (intPort.Test(out j))
        Console.WriteLine("Got value: " + j);
    else
        Console.WriteLine("No value!");
    // Final status
    Console.WriteLine(intPort.ToString());

    // Pause for user to read the output
    PressEnter();
}

void PressEnter()
{
    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.Write("Press Enter: ");
    Console.ResetColor();
    Console.ReadLine();
}
```

The `PressEnter` function is just a convenience to make the code pause. However, due to the asynchronous nature of the CCR, it does not always function as expected! In some examples, the “Press Enter” text will be interleaved with other text on the screen because the code is multi-threaded. (This is why it is displayed in yellow on the screen — so it stands out.)

When you run this code, you should see output similar to the following in the command window:

```
Posted a value
Port Summary:
  Hash:1424
  Type:System.Int32
  Elements:1
  ReceiveThunks:0
```

Receive Arbiter Hierarchy:

```
Got value: 42
Port Summary:
  Hash:1424
  Type:System.Int32
  Elements:0
  ReceiveThunks:0
Receive Arbiter Hierarchy:
```

```
No value!
Port Summary:
  Hash:1424
  Type:System.Int32
  Elements:0
  ReceiveThunks:0
Receive Arbiter Hierarchy:
```

Press Enter:

Notice that after posting to the port, the number of elements is 1 under `Port Summary`. After `Test` has executed, the number of elements is 0. On the second attempt to use `Test`, the code displays the message “No value!”

It is worth pointing out here that the number of `Receive Thunks` (a *thunk* is a chunk of receiver code — that is, the handler in a task) is zero in all cases, and nothing is listed under `Receive Arbiter Hierarchy`. This is because no receivers are specified in this example.

PortSets

A port can process only a single data type. Another class, called a `PortSet`, aggregates several different types of messages. In effect, it is a bunch of message queues that can all be treated as a single entity.

If you open `CCRExamplesTypes.cs`, you will find the definition of the *operations port* for the service, which contains three types of messages: `DsspDefaultLookup`, `DsspDefaultDrop`, and `Get`. (This is the minimum set of messages required for a service, explained in the next chapter.)

```
/// <summary>
/// CCRExamples Main Operations Port
/// </summary>
[ServicePort()]
public class CCRExamplesOperations : PortSet<DsspDefaultLookup,
    DsspDefaultDrop, Get>
{
}
```

Multiple messages of different types can be posted to a `PortSet`. Each message type can have a different handler, or you can create complex combinations of messages that must arrive to trigger a particular handler.

Note that it doesn't make sense to declare a `PortSet` with the same data type appearing in the list twice. If you have two possible sets of integer values, for example, you should use different enums or wrap them in different classes.

Limits on Declaring Portsets

Note that there are some inherent limitations to creating a `PortSet` declaratively. The CCR only allows up to 20 types in a `PortSet` for what is called the *desktop CLR*, i.e., the usual .NET environment. For the .NET Compact Framework (CF) environment, this is restricted to only eight because of a limitation in the Just-In-Time (JIT) compiler.

Novices sometimes run up against these limits because they add new operations to an existing service, thereby breaking it. It is particularly easy to exceed the limit for CF services.

However, there is a solution: Create the `PortSet` programmatically using `typeof`.

Looking again at the preceding example, the code can be rewritten as follows so that the `PortSet` is actually created at runtime as part of the class constructor:

```
/// <summary>
/// CCRExamples Main Operations Port
/// </summary>
[ServicePort()]
public class CCRExamplesOperations : PortSet
{
    public CCRExamplesOperations()
        : base(
            typeof(DsspDefaultLookup),
            typeof(DsspDefaultDrop),
            typeof(Get)
        )
    { }
}
```

However, there is still a problem with this approach. By creating the `PortSet` dynamically, there is no way to have the strong type-checking that is usually applied at compile-time when you try to post to a port.

The workaround is quite simple: Define overloads for the `Post` method for each of the different types (inside the `CCRExamplesOperations` class definition):

```
public void Post(DsspDefaultLookup msg)
{
    base.PostUnknownType(msg);
}
public void Post(DsspDefaultDrop msg)
{
    base.PostUnknownType(msg);
}
public void Post(Get msg)
{
    base.PostUnknownType(msg);
}
```

Notice that these overloads use the `PostUnknownType` method. This method looks up the list of acceptable types at runtime and posts to the associated port in the `PortSet`. If there is no match, then it throws an exception. You can also use `TryPostUnknownType` in your code, which returns a Boolean value indicating whether the post was successful or not, but doesn't cause an exception.

Having seen how posting messages works, it is now time to look at the other side — how messages are usually removed from a port.

Receivers and Arbiters

Strictly speaking, a port has a second queue, which is a list of *receivers* to be executed to retrieve and process messages. These receivers are sometimes called *continuations* and are conceptually similar to *callback procedures* in that they execute asynchronously when some event occurs — in this case the arrival of a message (or multiple messages). Much of the coding in MRDS services involves building receivers and the *handlers* that execute when messages are received.

You have already seen how messages are posted and can be extracted using `Test`. Now consider the case of a receiver. To use a receiver, you must construct it and then attach it to the port. This can be done most easily using wrappers that DSS and CCR provide:

```
// Create a Receiver and place it in the Dispatcher Queue
// This one is NOT persistent so it only fires once
Activate(
    Arbiter.Receive(false, intPort,
        delegate(int n)
            { Console.WriteLine("Receiver 1: " + n.ToString()); }
    )
);
```

The `Activate` method is defined in `DsspserviceBase` and wraps an `Arbiter.Receive` to use the default DSS dispatcher queue, which is `Environment.TaskQueue`. You can also create your own dispatcher if you want.

The `Arbiter.Receive` constructs a receiver using the following parameters:

- ❑ A persistent flag, which is `false` in the preceding code, i.e., non-persistent — it only receives once and then is removed
- ❑ A port called `intPort` to read from
- ❑ A delegate that is executed in-line

If the first parameter is `true`, then the receiver is persistent and will remain in the port's receiver list after processing a message (unless it is explicitly removed later). Many of the receivers that you create to control a robot need to be persistent.

The `intPort` is the same one declared in the section "Ports and Messages" earlier in the chapter, and it accepts integers as messages. Because of this, the delegate must expect to receive an integer as a parameter, which in this case is declared as `n`.

Chapter 2: Concurrency and Coordination Runtime (CCR)

You should realize by now that immediately after calling `Activate` execution continues with the next statement. The receiver quietly waits until an integer is posted to the `intPort`, or, if one has already been posted, then the receiver will fire straight away and process the value. Either way, it is quite likely that the receiver will execute on a different thread from the one that executed the `Activate`. This is an important concept, and you should be sure that you understand it.

Putting this in the context of a complete example, here is the code for the Receive example in `CCRExamples`:

```
// Receive a message on a port
void Receive()
{
    // Create a port
    Port<int> intPort = new Port<int>();

    // Create a Receiver and place it in the Dispatcher Queue
    // This one is NOT persistent so it only fires once
    Activate(
        Arbiter.Receive(false, intPort,
            delegate(int n)
                { Console.WriteLine("Receiver 1: " + n.ToString()); }
        )
    );

    // Add another receiver, but make it persistent
    Activate(
        Arbiter.Receive(true, intPort,
            delegate(int n)
                { Console.WriteLine("Receiver 2: " + n.ToString()); }
        )
    );

    Wait(100);
    Console.WriteLine("Receivers Activated:\n" + intPort.ToString());

    // Post a message
    // We could do this even before the receivers were
    // activated because it will just stay in the queue.
    // However, we want to see the effect on the receiver
    // queue in the port.
    intPort.Post(10);
    Wait(100);
    Console.WriteLine("After First Post:\n" + intPort.ToString());

    // Wait a while then post another message
    Wait(100);
    intPort.Post(-2);
    // The Receiver is still active so wait for it to process
    // the second message as well
    Wait(100);
    // And finally a third message ...
    intPort.Post(123);
    Wait(100);
}
```

Chapter 2: Concurrency and Coordination Runtime (CCR)

The preceding example creates two receivers: the first is nonpersistent and the second is persistent. Then it posts three messages to the port. It shows what happens to the state of the port by displaying port summaries. The output is shown here:

```
Receivers Activated:
Port Summary:
  Hash:1424
  Type:System.Int32
  Elements:0
  ReceiveThunks:2
Receive Arbiter Hierarchy:
Receiver`1(Onetime) with method unknown:<Receive>b__10 nested under
  none
Receiver`1(Persistent) with method unknown:<Receive>b__11 nested under
  none

Receiver 1: 10
After First Post:
Port Summary:
  Hash:1424
  Type:System.Int32
  Elements:0
  ReceiveThunks:1
Receive Arbiter Hierarchy:
Receiver`1(Persistent) with method unknown:<Receive>b__11 nested under
  none

Receiver 2: -2
Receiver 2: 123
```

In the first port summary, there are two receivers in the receiver list. After Receiver 1 fires, there is only one receiver left. This second receiver is persistent, as you can see from the two messages that it displays in response to integers being posted to the port.

Persisted handlers are vital to service-oriented applications. Consider a web server. It waits for HTTP requests to come in on a port (this is TCP/IP port 80, which is not the same as a CCR port) and then in response to this request it sends back a web page. This cycle of request/response (possibly with an error) is fundamental to services. When you learn about the DSS in the next chapter, you will see that this is the pattern of what is called a *service operation*.

Choice

The `Choice` arbiter, which you can think of as a logical OR, waits on two receivers until one of them fires. It then shuts down the unused receiver.

One common use of `Choice` is to handle responses from a service. The response can be either the requested data or an exception, which is returned as a `SOAP Fault`. In this case, the `Choice` has two receivers that correspond to success or failure of the operation.

Chapter 2: Concurrency and Coordination Runtime (CCR)

The following example from `CCRExamples` shows how `Choice` works. You can run it by selecting the `Choice` example from the menu.

```
// Choice -- Choose the first message to arrive (Logical OR)
void Choice()
{
    // Create a PortSet that takes two different data types
    PortSet<bool, int> ps = new PortSet<bool, int>();

    // If you post the messages BEFORE setting up the Choice,
    // you might see a different result compared to posting
    // them AFTER it has been created
    //ps.Post(1000000);
    //ps.Post(true);

    // Create the Choice and activate it
    Activate(
        Arbiter.Choice<bool, int>(ps,
            // Create delegates for each type in the PortSet
            delegate(bool b)
            { Console.WriteLine("Choice: " + b.ToString()); },
            delegate(int n)
            { Console.WriteLine("Choice: " + n.ToString()); }
        )
    );

    // Post two messages - Only one will be selected
    // NOTE: If you run this often enough, you might see
    // cases where either of these is displayed because the
    // result is indeterminate if there are messages of both
    // types in the port when the Choice executes
    ps.Post(1000000);
    ps.Post(true);
}
```

When you run this code, one value displays. As the preceding code shows, the usual result is `1000000`, but it is possible for a value of `true` to be displayed depending on subtle timing variations in the CCR. If you uncomment the `Post` instructions at the top of the routine and remove the comments at the bottom, you will most likely find that the behavior is reversed. `Choice` does not guarantee which of the ports it will choose from if two messages are waiting when it is evaluated.

In the V 1.5 Refresh, Microsoft introduced a new concise syntax for using `Choice` that can be used inside iterators. As you should understand by now, the `yield return` statement can wait on a message to be received. Many of the operations you can perform on services are declared so that they return a `PortSet`. For example, saving the state of a service (covered in Chapter 3) can now be done as follows:

```
yield return (Choice)SaveState(_state);
```

The `SaveState` method returns a `PortSet` that can receive either a `Default Replace Response` or a `Fault`. You should really check to ensure that it was successful — i.e., did not return a `Fault` — it is unusual for this to fail in a service that has been properly debugged. However, this single line of code does suspend execution until the `SaveState` has completed, which is important if you are trying to save the state as part of the shutdown process in your service's `Drop` handler, for example.

Chapter 2: Concurrency and Coordination Runtime (CCR)

A slightly longer example shows another way to use it:

```
PortSet<SuccessResult, Exception> successResultPort = SomeFunction();

yield return (Choice)successResultPort;
SuccessResult s = successResultPort;
if (s == null)
    Console.WriteLine("Exception: " + (Exception)successResultPort);
```

Success/Failure ports are discussed later in the chapter under the section “Error Handling,” but it suffices to say that `SomeFunction` returns a `PortSet` that can have two possible results: `SuccessResult` or `Exception`. Presumably the function kicks off some tasks that will eventually post a result. This use of `Choice` does not require delegates and is much cleaner.

The example also demonstrates how a message can be extracted from a `PortSet` implicitly. When the variable `s` is initialized, it receives a `SuccessResult` message if the `PortSet` is holding one. (You know that there is some type of message in the `PortSet` because of the preceding `Choice`.) If there is no `SuccessResult`, then the code grabs the `Exception` from the `PortSet` and displays it.

Note that `Choice` does not persist. In fact, it performs a *teardown* as soon as one of the receivers executes. This actually prevents a second message from arriving, although in the binary case with only two message types it is hard to see how both types of message could arrive unless there was a bug in the code. The `Choice` class can handle an arbitrary number of receivers, but you will most commonly see it with only two.

Join

Similar to a logical AND, a `JoinReceive` arbiter waits for two receivers to complete before continuing. In terms of synchronization, this allows a “rendezvous” point in the code where two tasks must both complete before the rest of the code is executed.

Consider the `Join` example from `CCRExamples`:

```
// Join -- Wait for two messages (Logical AND)
void Join()
{
    // Set up three different ports
    Port<bool> p1 = new Port<bool>();
    Port<int> p2 = new Port<int>();
    Port<string> p3 = new Port<string>();

    // Join on ports p1 and p2
    Arbiter.Activate(Environment.TaskQueue,
        Arbiter.JoinedReceive(
            false, p1, p2,
            delegate(bool b, int i)
            {
                Console.WriteLine("Join 1: {0} {1}", b, i);
                // Now post to p2 so the other Join can complete
```

(continued)

Chapter 2: Concurrency and Coordination Runtime (CCR)

(continued)

```
        p2.Post(i+1);
    }
    )
);

// Join on ports p2 and p3
Arbiter.Activate(Environment.TaskQueue,
    Arbiter.JoinedReceive(
        false, p2, p3,
        delegate(int i, string s)
        {
            Console.WriteLine("Join 2: {0} {1}", i, s);
            // Now post to p2 so the other Join can complete
            p2.Post(i-1);
        }
    )
);

// Now post to the ports
// NOTE: It is not possible to tell which Join will be
// executed because it depends how quickly the messages
// arrive. In general, all three messages would not be
// sent at the "same time" as in the code below.

p1.Post(true);
p3.Post("hello");
p2.Post(99);
}
```

When you run this example, the output might look like the following:

```
Join 2: 99 hello
Join 1: True 98
```

Alternately, the result might be as follows:

```
Join 1: True 99
Join 2: 100 hello
```

Try executing the example repeatedly; you should see both behaviors. It all depends on where the CCR is up to in its cycle of evaluating receivers for execution (because three messages arrive almost simultaneously); it is therefore nondeterministic.

Notice here that there is contention for messages on port `p2`. However, both of the `Joins` eventually run as long as enough messages are posted.

Combining Arbiters

Arbiters can be nested. This means that you can place a `Choice` inside a `Join`, or vice versa, which enables you to create arbitrarily complex logic. This chapter does not provide any examples of this, but you can play around with it on your own by combining the simple examples shown previously.

Bear in mind that a `Choice` atomically removes all nested arbiters from their ports once it is satisfied. Consider this when designing your logic.

Receiving Multiple Messages

A common requirement is to accumulate a number of messages before proceeding to execute the next step in the code. The CCR provides two different ways to do this:

- ❑ Multiple Item Receive
- ❑ Multiple Port Receive

The `Join` discussed previously is considered *static* because it only takes two ports defined at compile-time. You can also create a *dynamic* `Join` by waiting for multiple messages to arrive on a port.

The following example shows how to read six messages from a port in one atomic operation. This is a persistent receiver, so it groups messages together in clumps of six items and displays them:

```
// Receive multiple messages of the same type
void MultipleItemReceive()
{
    Port<int> p = new Port<int>();

    Arbiter.Activate(Environment.TaskQueue,
        Arbiter.MultipleItemReceive(
            true, p, 6,
            delegate(int[] array)
            {
                string s = "";
                for (int i = 0; i < array.Length; i++)
                    s += array[i].ToString() + " ";
                Console.WriteLine("{0} Items: {1}", array.Length, s);
            }
        )
    );

    for (int i = 0; i < 4; ++i)
        p.Post(i + 1);
    Wait(100);

    Console.WriteLine(p.ToString());

    for (int i = 0; i < 4; ++i)
        p.Post(i + 1);
    Wait(100);

    Console.WriteLine(p.ToString());

    for (int i = 0; i < 4; ++i)
        p.Post(i + 1);
    Wait(100);

    Console.WriteLine(p.ToString());
}
```

Chapter 2: Concurrency and Coordination Runtime (CCR)

Notice that the code posts messages in three batches of four at a time (for a total of 12 messages), so the receiver fires partway through the second batch. The output looks like the following if you run the `MultipleItemReceive` example in `CCRExamples`:

```
Port Summary:
  Hash:1424
  Type:System.Int32
  Elements:4
  ReceiveThunks:1
Receive Arbiter Hierarchy:
JoinSinglePortReceiver(Persistent) with method unknown: ~
<MultipleItemReceive>b__1c nested under
  none

6 Items: 1 2 3 4 1 2
Port Summary:
  Hash:1424
  Type:System.Int32
  Elements:2
  ReceiveThunks:1
Receive Arbiter Hierarchy:
JoinSinglePortReceiver(Persistent) with method unknown: ~
<MultipleItemReceive>b__1c nested under
  none

6 Items: 3 4 1 2 3 4
Port Summary:
  Hash:1424
  Type:System.Int32
  Elements:0
  ReceiveThunks:1
Receive Arbiter Hierarchy:
JoinSinglePortReceiver(Persistent) with method unknown: ~
<MultipleItemReceive>b__1c nested under
  none
```

Three port summaries are displayed so that you can see what is happening. The first summary shows that there are four messages (elements) in the port and one receiver (thunk), which is a `MultipleItemReceiver`. Another four messages are posted and, before the second port summary can be displayed, the receiver outputs the values of six messages. Two items are left when the second summary is displayed. The last four messages are posted, and the receiver picks them up for output, and then the port summary shows that there are no messages remaining to be processed.

The `MultipleItemReceiver` can only receive from a single port, unlike the `JoinReceiver` shown earlier. If you want to use different ports, then you need to use a `MultiplePortReceiver` instead.

An example of the `MultiplePortReceiver` is also included in `CCRExamples`. It looks like this:

```
// Receive multiple messages of the different types
void MultiplePortReceive()
{
    // Create a port set that accepts two different data types
```

Chapter 2: Concurrency and Coordination Runtime (CCR)

```
PortSet<int, double> pSet = new PortSet<int, double>();

//Arbiter.Activate(Environment.TaskQueue,
Activate(
    Arbiter.MultipleItemReceive(
        pSet, 10,
        delegate(ICollection<int> colInts, ICollection<double> colDoubles)
        {
            Console.WriteLine("Ints: ");
            foreach (int i in colInts)
                Console.WriteLine(i + " ");
            Console.WriteLine();
            Console.WriteLine("Doubles: ");
            foreach (double d in colDoubles)
                Console.WriteLine("{0:F2} ", d);
            Console.WriteLine();
        }
    )
);

// Generate some random numbers and post them using
// different message types depending on their values
Random rnd = new Random();
for (int i = 0; i < 10; ++i)
{
    double num = rnd.NextDouble();
    if (num < 0.5)
        pSet.Post((int)(num * 100));
    else
        pSet.Post(num);
}
}
```

Although this is referred to as a Multiple Port Receiver, it actually uses the `MultipleItemReceiver` but with a `PortSet`. This particular `PortSet` accepts integers and doubles. Notice that the delegate has two parameters, which are collections of the corresponding types. The piece of code at the bottom of the example randomly generates a set of integers and doubles.

Every time you run the example, you will get a different result. Two examples of the output are shown here:

```
Ints: 36 25 15
Doubles: 0.74 0.71 0.87 0.81 0.51 0.51 0.79
Ints: 23 17 34 18 45 40
Doubles: 0.53 0.56 0.96 0.59
```

Note that there are different numbers of items in each of the collections, but the total number of items is 10 in both cases.

Interleave

Services typically offer a large number of different operations. As you will see in Chapter 3, these are defined as a set of classes, called *operations*, which are the message types you can send to a service. Each of these message types needs to have a corresponding receiver.

Chapter 2: Concurrency and Coordination Runtime (CCR)

Setting up multiple receivers is the job of an `Interleave`. It also provides important facilities for protecting resources and housekeeping by using three groups of receivers: Tear Down, Exclusive and Concurrent:

- ❑ The Tear Down group contains receivers that should be called when the `Interleave` is supposed to shut down. The processing of messages stops as soon as a Tear Down receiver is executed; and once it completes, the entire `Interleave` is disposed, effectively cancelling all of the receivers. Teardown messages take priority.
- ❑ The Concurrent receiver group is easy to understand: It operates as if you had simply set up a bunch of receivers yourself. They can all run in parallel (if there are enough threads).
- ❑ The Exclusive receiver group is the key, however. It only allows a single receiver to execute at a time. It waits for an executing Concurrent receiver to finish before starting an Exclusive receiver. If more Exclusive receivers are triggered before the first one finishes, then they are queued and executed one after another. Concurrent receivers cannot execute while an Exclusive receiver is running.

There is a subtle issue here in that the actual messages are held by the `Interleave`, which is said to be *guarding* the ports. You will not see the messages queued in the individual ports in a `PortSet` if it is controlled by an `Interleave`. They are released one at a time if they are Exclusive and remain in the pending exclusive requests queue in the `Interleave`.

Exclusivity is an important concept for services. In particular, services usually contain an internal *state*. This state contains information that controls the operation of the service, and it must remain consistent. If several threads updated the state at the same time, there is no telling what might happen. Therefore, any handler that wants to update the state should be Exclusive.

Reading the state, however, can be done concurrently because a Concurrent handler can never run at the same time as an Exclusive handler, and it will therefore always see a consistent view of the state.

An `Interleave` persists in the sense that it contains a set of receivers. However, the individual receivers in an `Interleave` can be persistent or nonpersistent. In general they are persistent. Notice in the following example that the `DropHandler` in the `TeardownReceiverGroup` is nonpersistent. It wouldn't make sense for this to be persistent because once you drop a service, it is gone!

```
Activate(
    Arbiter.Interleave(
        new TeardownReceiverGroup(
            Arbiter.Receive<DsspDefaultDrop>(false, _mainPort, DropHandler)
        ),
        new ExclusiveReceiverGroup(
            Arbiter.Receive<LaserRangeFinderResetUpdate>(true, _mainPort, ↵
LaserRangeFinderResetUpdateHandler),
            Arbiter.Receive<LaserRangeFinderUpdate>(true, _mainPort, ↵
LaserRangeFinderUpdateHandler),
            Arbiter.Receive<BumpersUpdate>(true, _mainPort, ↵
BumpersUpdateHandler),
            Arbiter.Receive<BumperUpdate>(true, _mainPort, ↵
BumperUpdateHandler),
            Arbiter.Receive<DriveUpdate>(true, _mainPort, ↵
DriveUpdateHandler),
```

```
        Arbiter.Receive<WatchDogUpdate>(true, _mainPort, ◀  
WatchDogUpdateHandler)  
    ),  
    new ConcurrentReceiverGroup(  
        Arbiter.Receive<Get>(true, _mainPort, GetHandler),  
        Arbiter.Receive<dssp.DsspDefaultLookup>(true, _mainPort, ◀  
DefaultLookupHandler)  
    )  
);
```

Services sometimes use one `Interleave` during service initialization, and then a different `Interleave` once they are fully up and running. In this case, the receivers in the initialization `Interleave` might not be persistent.

Alternatively, once you have set up an `Interleave`, you can add to it using its `CombineWith` method. The `DsspServiceBase` class automatically creates a `MainPortInterleave` for you. The following code shows how some receivers can be added without affecting existing receivers. Note that no receivers are specified in this case for the `TeardownReceiverGroup` or the `ConcurrentReceiverGroup` but you still have to supply a parameter in this construct.

```
    MainPortInterleave.CombineWith(  
        new Interleave(  
            new TeardownReceiverGroup(),  
            new ExclusiveReceiverGroup(  
                Arbiter.Receive<simengine.InsertSimulationEntity>(true, ◀  
_notificationTarget, InsertEntityNotificationHandler),  
                Arbiter.Receive<simengine.DeleteSimulationEntity>(true, ◀  
_notificationTarget, DeleteEntityNotificationHandler),  
                Arbiter.Receive<FromWinformMsg>(true, _fromWinformPort, ◀  
OnWinformMessageHandler)  
            ),  
            new ConcurrentReceiverGroup()  
        )  
    );
```

As a final example, consider the case where you have a sequence of tasks that must be executed as a unit and not preempted. You can add them all as `Exclusive` receivers. The first one in this multi-step process is executed when a new message arrives. Just before it completes, it posts a message to the second one, and so on. Each task passes control to the next one. No `Concurrent` receivers can execute while an `Exclusive` receiver is ready to run. However, other `Exclusive` tasks can sneak into the queue.

Dispatchers and Dispatcher Queues

Dispatchers and dispatcher queues enable you to create as many thread pools as you like and to assign different priorities and scheduling policies. This is in stark contrast to the .NET CLR, which operates with a single thread pool (if you use `System.Threading`).

A dispatcher manages a pool of threads, and it can have multiple dispatcher queues feeding into it. The items in a dispatcher queue are (usually) the combination of a handler and some data from a port. Although dispatchers can handle any number of dispatcher queues, quite often a dispatcher will only have one queue.

Chapter 2: Concurrency and Coordination Runtime (CCR)

When you create a DSS service, the base class (`DssServiceBase`) creates a default dispatcher and dispatcher queue for you. Consequently, you do not usually have to worry about creating these objects or managing them when you are writing a DSS service.

The number of threads managed by a dispatcher can be specified at construction time. Unlike the CLR thread pool, this pool of threads will not automatically grow or shrink during the lifetime of the dispatcher. (This helps to make the CCR more efficient but at the cost of possible bottlenecks if you execute too many long-running tasks and soak up all of the threads.)

The following code shows the constructor signatures for the dispatcher class obtained using reflection. You can do this for any of the classes in MRDS by typing the class name into your code, right-clicking it with the mouse, and selecting Go To Definition from the pop-up menu. This is a useful feature.

Alternatively, you can use the Object Browser.

```
namespace Microsoft.Ccr.Core
{
    public sealed class Dispatcher : IDisposable
    {
        public Dispatcher(int threadCount, string threadPoolName);
        public Dispatcher(int threadCount, ThreadPriority priority,
bool useBackgroundThreads, string threadPoolName);
        public Dispatcher(int threadCount, ThreadPriority priority,
DispatcherOptions options, string threadPoolName);
        public Dispatcher(int threadCount, ThreadPriority priority,
DispatcherOptions options, ApartmentState threadApartmentState,
string threadPoolName);
        ...
    }
}
```

By default, the number of threads is the number of CPUs, or CPU cores if it is a multi-core CPU; but the minimum number of threads is two, even for a single CPU. (You get the default number of threads when you specify zero in the constructor.)

If you have a resource that must only be accessed by one thread at a time, then you can use just a single thread in a dispatcher specifically for this purpose. Because there is only one thread, it doesn't matter how many tasks are queued — only one task can execute at a time. This is an issue, for example, if you have legacy code that uses a Single-Threaded Apartment (STA) model. Windows Forms is a case in point, and running forms requires special consideration, a subject covered in Chapter 4.

Dispatchers and dispatcher queues are relatively lightweight, so there is no real penalty in having several of them. However, it is unlikely that you will need more than one for most applications.

One scenario in which you might want multiple dispatchers is when you want to set different thread priorities. All threads in a dispatcher have the same priority, but you can have multiple dispatchers. `ThreadPriority` is an enum in the `System.Threading` namespace. You can find the values and their descriptions using IntelliSense, but for ease of reference they are listed in the following table:

Thread Priority Values

Value	Description
Lowest	Threads can be scheduled after threads with any other priority.
BelowNormal	Threads can be scheduled after threads with Normal priority but before threads with Lowest priority.
Normal	Threads can be scheduled after threads with AboveNormal priority but before threads with BelowNormal priority. (This is the default.)
AboveNormal	Threads can be scheduled before threads with Normal priority but after threads with Highest priority.
Highest	Threads can be scheduled before threads with any other priority.

The `threadPoolName` is useful if you use the Threads window of the Visual Studio Debugger because you can use it to identify threads from particular dispatchers.

There are a few other properties defined for dispatchers that you might use occasionally:

- `WorkerThreadCount` can be used to examine or change the number of threads in the pool.
- `ProcessedTaskCount` tells you how many tasks are queued for processing.
- `PendingTaskCount` keeps track of how many tasks have been executed since the dispatcher was created.

A dispatcher queue is used to queue tasks. If you are wondering why this functionality is not integrated into the dispatcher, it's because this enables queues to be defined with separate scheduling policies. These policies are defined in an enum called `TaskExecutionPolicy`. Descriptions of each of the policies are given in the following table and should be self-explanatory. `Unconstrained` is the default policy. The other policies involve either queue depth or scheduling rate, and there are two possibilities: discard tasks or force threads attempting to submit new tasks to wait.

Dispatcher Task Execution Policy Values

Value	Description
<code>Unconstrained</code>	All tasks are queued with no constraints (the default).
<code>ConstrainQueueDepthDiscardTasks</code>	Tasks enqueued after the maximum depth are discarded.
<code>ConstrainQueueDepthThrottleExecution</code>	Maximum depth is enforced by putting posting threads to sleep until the queue depth falls below the limit.

Table continued on following page

Value	Description
ConstrainSchedulingRateDiscardTasks	Tasks enqueued while the average scheduling rate is above the limit are discarded.
ConstrainSchedulingRateThrottleExecution	Once the average scheduling rate exceeds the limit, posting threads are forced to sleep until the rate falls below the limit.

Implementing Common Control Structures

Most programmers look for “patterns” or templates that they can use in their coding. This section provides a number of patterns that are commonly used in MRDS programming.

Sequential Processing

Executing a sequence of operations is quite easy with an iterator, as you have already seen. However, sometimes you might want to spin off a separate task but you need to wait for it to finish. You can take two approaches to this:

- ❑ Create a completion port to wait on and have the other task send a message when it is finished.
- ❑ Use the `ExecuteToCompletion` arbiter.

Using a completion port that you create yourself is quite easy and you should be able to do that at this point. It doesn’t matter what type of port it is because you will only send a single message to it to indicate to the first task that the second task has finished. In fact, you can set up a string of receivers, with each of them referring to the next handler in the sequence. The handlers can “daisy chain” from one to another by sending a message to the completion port when they have finished. This multi-step process can call a mixture of simple handlers or iterators.

The `ExecuteToCompletion` arbiter works as shown in the next example. A parent iterator yields to the `ExecuteToCompletion` arbiter to run a child iterator. In the following code, a new iterator task is explicitly created. You could also use `Arbiter.FromIteratorHandler`, discussed previously. Notice that in this case a `double` is passed to the iterator.

```
private IEnumerator<ITask> Parent()
{
    // Do some work
    ...

    // Now call the child
    Console.WriteLine("Yielding to child");
    yield return Arbiter.ExecuteToCompletion(
        taskQueue,
        new IterativeTask<double>(3.1415926f, Child)
    );
}
```

```
        Console.WriteLine("Child completed");

        // Carry on with our own processing
        ...
    }

private IEnumerator<ITask> Child(double number)
{
    // Execute some code
    ...
    // Yield now
    yield return Arbiter.xxx();

    // Execute some more code
    ...
    // Yield again
    yield return Arbiter.xxx();

    // And so on
    ...
}
```

The `Parent` function calls the `Child`, which executes a series of steps. The CCR knows when an iterator is complete, either by reaching the end of the function or via `yield break`. This terminates the iteration, and the original `yield return` in the `Parent` is satisfied.

Note that the `Child` does not have to be an iterator — you could use `Arbiter.FromHandler` to create the task. In this case, the `yield return` in the parent is similar to a normal procedure call except that it provides an opportunity for the CCR to do some scheduling. This helps to break long-running tasks up into bite-sized pieces and share the CPU.

Scatter/Gather

The term “Scatter/Gather” refers to sending out multiple requests and then waiting for all of the responses to arrive. Sending the requests is the easy part — you just need to use `Post`. Then you need to wait for the responses from the various operations. Depending on your logic, you might want to wait until one of the operations has completed, all of them have completed, or any combination in between.

The `Choice` arbiter can wait on several receivers. Examples typically focus on only two options because in most cases you are only interested in getting a response or a fault. This particular form of the `Choice` uses a `PortSet` that has only two possible message types. However, an alternate form of `Choice` accepts an array of receivers to which you can add as many receivers as you like. This enables you to wait for one of many responses.

To wait for all operations to complete, you can use a `Multiple Item Receiver` or a `Multiple Port Receiver`. It is possible, for example, to send several requests to the same port, in which case you need to receive multiple items. However, it is more likely that you will want to receive from multiple different ports.

Chapter 2: Concurrency and Coordination Runtime (CCR)

Although it is a little advanced for this chapter, consider the following code from the Synchronized Dance service in Chapter 14 that controls two robots simultaneously:

```
PortSet<DefaultUpdateResponseType, Fault> p =
new PortSet<DefaultUpdateResponseType, Fault>();
drive.RotateDegrees[] rotateRequests =
new drive.RotateDegrees[_drivePorts.Count];

for (i = 0; i < _drivePorts.Count; i++)
{
    rotateRequests[i] = new drive.RotateDegrees();
    rotateRequests[i].Body.Degrees = _state.RotateAngle;
    rotateRequests[i].Body.Power = _state.RotatePower;
    rotateRequests[i].ResponsePort = p;
    _drivePorts[i].Post(rotateRequests[i]);
}

yield return Arbiter.MultipleItemReceive(p, drivePorts.Count,
driveHandler);
```

The code creates a request for each robot and then sends it in the `for` loop. The `ResponsePort` of each request is set to the same `PortSet`, `p`. Then the code waits on a response from each robot using the `MultipleItemReceive`. Note that it is waiting on a `PortSet` and the response can be either the default response or a fault for each robot, but there is only one message per robot.

As another example, if your initialization consists of three steps that can execute in parallel, you can spawn the three steps and have each step post a message to a completion port. The startup procedure must wait for all three messages to arrive on the completion port.

State Machines

There is nothing special about finite state machines (FSMs); they are often used in robotics applications. The `SimMagellan` example in Chapter 7 and the `ExplorerSim` example in Chapter 9 use state machines. Some of the MRDS samples also use state machines. Because this is a fundamental concept in computer science, it is not discussed here.

Last Message

Sometimes messages arrive fast and furious. You might not want to process all of these messages, especially if it takes a long time to process a single message.

You have seen that it is possible to check the `ItemCount` on a port. Later, in Chapter 9, you will encounter the `ExplorerSim` example, where the robot is processing laser range finder (LRF) information. If it cannot keep up with the data, it needs to throw away some readings and only process the latest message.

The following code fragment shows how `ExplorerSim` discards old LRF messages:

```
/// <summary>
/// Gets the most recent laser notification. Older notifications are dropped.
/// </summary>
/// <param name="laserData">last known laser data</param>
```

```
/// <returns>most recent laser data</returns>
private sicklrf.State GetMostRecentLaserNotification(sicklrf.State laserData)
{
    sicklrf.Replace testReplace;
    Port<sicklrf.Replace> laserPort = _laserNotify;

    int count = laserPort.ItemCount;

    for (int i = 0; i < count; i++)
    {
        testReplace = (sicklrf.Replace)laserPort.Test();
        if ((testReplace != null) && (testReplace.Body.TimeStamp >
laserData.TimeStamp))
        {
            laserData = testReplace.Body;
        }
    }
    ...
}
```

Notice that this code uses a particular syntax to obtain the `laserPort` from the `PortSet` for laser notifications. This implicit assignment operator is automatically generated by `DssProxy` when it creates the Proxy DLL, and you do not need to worry about it.

When you are using a joystick to drive a robot, there can be a large number of requests to change the drive power in a short period of time. Unfortunately, in this case, the `SetDrivePower` handler is normally set up as an Exclusive receiver as part of the `MainPortInterleave` that DSS creates for you.

This is jumping ahead to the next chapter, but be aware that this is a trap for beginners. You cannot get the count of waiting messages from the `SetDrivePower` port in this case because the messages are held up in the `Interleave` and the queue length on the `SetDrivePower` port will always appear to be zero!

In this case, you must extract messages from the `MainPortInterleave`. You can check how many messages are pending using the `PendingConcurrentCount` and `PendingExclusiveCount` properties, and then you can extract messages using the `TryDequeuePendingTask` method. Alternatively, the handler can post all incoming messages to an internal port that only processes one message at a time, and the internal handler can check the queue length on the internal port. This approach is used in Chapter 17, where services are developed for new robots.

Time Delays

Perhaps you are familiar with `System.Threading.Thread.Sleep`. Although you can use `Thread.Sleep` in a CCR environment, this is *not* the recommended way to introduce a delay.

In the preceding sample code you might have noticed calls to `Wait` to suspend execution of the code temporarily. This is a convenience routine that is defined in the `CCRExamples` service. It simply calls `Thread.Sleep`.

Chapter 2: Concurrency and Coordination Runtime (CCR)

If you look in the code, another `Wait` method is commented out that uses the `TimeoutPort`, defined in the `CcrServiceBase` class:

```
/// <summary>
/// Wait for a specified period of time
/// </summary>
/// <param name="millisec">Delay time in milliseconds</param>
/// <remarks>Suspends execution without using Thread.Sleep, but creates
a deadlock.</remarks>
void Wait(int millisec)
{
    // Create OS event used for signalling.
    // By creating a new one every time, you can call this routine
    // multiple times simultaneously because there is no shared resource.
    AutoResetEvent signal = new AutoResetEvent(false);

    // Schedule a CCR Timeout task that will execute in parallel with
    // this method and signal when the Timeout has completed.
    Activate(
        Arbiter.Receive(
            false,
            TimeoutPort(millisec),
            delegate(DateTime timeout)
            {
                // Done. Signal so that execution can continue.
                signal.Set();
            }
        )
    );

    // Block until Timeout completes
    signal.WaitOne();
}
```

The receiver waits for the specified time delay in milliseconds. (There is also an overload for `TimeoutPort` that uses a `TimeSpan`.) It is not a persistent receiver (the first parameter is `false`).

Notice that the delegate does nothing except set the signal because the objective is just to suspend execution for some length of time. The last line of the routine blocks execution and waits for the signal from the delegate.

Unfortunately, when there are only two threads, this can lead to a deadlock. To test this, uncomment the code for `Wait` and comment out the original (one-line) `Wait` method. Make sure that you have commented out the `ActivationSettings` attribute at the top of the code so you don't have six threads. Now recompile the code.

If you are lucky enough to have a quad-processor PC, then you should not comment out `ActivationSettings` but instead change the number of threads to two. Otherwise, you will get four threads by default and you won't see the deadlock.

Run the program and select the `Spawn` example, which is option number 1. What happens? A start message appears and then you are left staring at a blinking cursor. Don't stare for too long because nothing is going to change.

It appears that this fancy version of `Wait` is not a solution to the problem of blocking a thread — it still has the same effect as `Thread.Sleep`. The subtle issue here is that if `Wait` is called while another `Wait` is active, then two threads are blocked and there are no threads left to execute the delegates and wake up the waiting threads. The main loop calls `Wait` after executing your selection, and then the `Spawn` example also executes `Wait`.

You must be very careful not to deadlock the CCR by soaking up all of the threads. If you know that one of your threads will block, and there is no easy way around it, then you can explicitly specify a separate dispatcher and the number of threads to use on your service.

For reviewing this chapter, many thanks to George Chrysanthakopoulos, lead developer for MRDS, who says that “yield is your friend.”

Inside an iterator, it is very easy to create a delay directly by using the `TimeoutPort` in a `yield return`:

```
// Wait for some time
yield return Arbiter.Receive(
    false,
    TimeoutPort(timeDelay),
    delegate(DateTime timeout) {}
);
```

Timers set up in this way are not very reliable because they use CLR timers. There is a fair amount of variation in the time intervals that elapse. This should be corrected with new timers in version 2.0 of MRDS. In the meantime, you can search the forum for ways to work around this problem if you need highly accurate timers or timers for very short time periods. You need to use a `CcrStopwatch` with V1.5. You can find some threads on this topic in the MRDS discussion forum.

Periodic Events

Executing code periodically is simply a matter of using the `TimeoutPort` with the `persist` parameter set to `true` and specifying the handler to call. Note that the handler must accept a `DateTime` parameter. This approach results in the handler being called repeatedly. If the handler cannot perform its processing within the specified time period, then the messages back up and the handler will be flat-out running all the time.

Another approach that is probably better is for the handler to call itself back when it has finished processing. For example, the following handler activates a new receiver to call itself as the last statement in the routine, but it is a nonpersistent receiver:

```
static int TimerCounter = 0;

// A handler that calls itself periodically
void PeriodicTimerExample(DateTime dt)
{
    int timeDelay = 1000;

    TimerCounter++;
    if (TimerCounter > 10)
```

(continued)

Chapter 2: Concurrency and Coordination Runtime (CCR)

(continued)

```
        return;

        Console.WriteLine("Tick {0} ...", TimerCounter);

        // Wait for some time
        Activate(Arbiter.Receive(
            false,
            TimeoutPort(timeDelay),
            PeriodicTimerExample));
    }
}
```

To get the process started, you must set up an initial receiver. In the following code, the timeout is set to only 10 milliseconds because this is just to get the process going. Note also that the `TimerCounter` is reset to zero first because the timer handler should only run 10 times:

```
// Reset the counter and kick off the timer
TimerCounter = 0;
Activate(Arbiter.Receive(false, TimeoutPort(10), PeriodicTimerExample));
```

When you run this code by selecting Periodic Timer in `CCRExamples`, the output might look like the following:

```
Tick 1 ...
Select from the following:
0 = Exit
1 = Spawn
2 = Task From Handler
3 = Task From Iterator
4 = SpawnIterator
5 = Post and Test
6 = Receive
7 = Choice
8 = Join
9 = MultipleItemReceive
10 = MultiplePortReceive
11 = Periodic Timer
12 = Causality
Enter a number: Tick 2 ...
Tick 3 ...
Tick 4 ...
Tick 5 ...
Tick 6 ...
Tick 7 ...
Tick 8 ...
Tick 9 ...
Tick 10 ...
```

This might not seem very exciting, but you can make another selection from the menu while the timer is ticking. See what happens then! Do you run out of threads? If so, uncomment the `ActivationSettings` to give yourself six threads and try again.

You can improve the accuracy of a periodic timer by using the parameter, which is a timestamp. If the code remembers the timestamp from the previous invocation, it can subtract the two and figure out the amount of time that actually elapsed. The new delay for the next timeout can be adjusted so that the timer fires on schedule the next time. At least that is the theory — it isn't always reliable.

Setting Limits with Timeouts

From time to time, you might want to execute an operation that might not succeed but for which there is no indication of failure. For example, if you send a request to a service that is hung, then you will never receive a reply. In this case, you want to use a timeout so that you don't end up waiting forever.

Consider the following code fragment from the Lynx6Arm simulation in Chapter 8:

```
// Wait for notification or time out if nothing is received
yield return Arbiter.Choice(
    Arbiter.Receive(false,
        TimeoutPort(DsspOperation.DefaultLongTimeSpan),
        delegate(DateTime dt)
        { LogError("Timeout waiting for visual entity"); Shutdown(); }),
    Arbiter.Receive<simengine.InsertSimulationEntity>(false,
        notificationTarget,
        delegate(simengine.InsertSimulationEntity ins)
        {
            _entity = (Lynx6ArmEntity)ins.Body;
        })
);
```

The `Choice` waits on the `TimeoutPort` and the Simulator notification port. If the simulation entity is successfully created, there is no problem. However, if it is not, the timeout will kick in.

Note that all DSS operations have a `TimeSpan` property that you can set before you post a request. This specifies the maximum amount of time to wait for a reply. If there is no response within that time period, then a `Fault` is generated automatically by DSS. This is an alternative approach.

Asynchronous I/O

Most of the robot implementations provided with MRDS use a serial port for communication. The .NET environment makes programming easy with the `SerialPort` class. However, serial ports normally operate asynchronously. It is not a good idea to tie up a CCR thread waiting for a read on a serial port, but sometimes this is your only option. (For example, look at the Hemisson robot in Chapter 17.)

Covering how to use a serial port is outside the scope of this chapter. You should look at some of the sample code in MRDS and also study Chapter 17. Be aware, however, that there are many different ways to implement serial communications.

The best example is probably in the Sick Laser Range Finder code. You can find this service in `samples\Sensors\SickLRF` and you should look at `SerialIOManager.cs`, which defines a set of operations that it can perform (`Open`, `Close`, `SetRate`, `Send`) and has a public `ResponsePort` to which it posts received packets. External code can wait on the `ResponsePort` for messages, which will be either a `Packet` or an `Exception`.

Chapter 2: Concurrency and Coordination Runtime (CCR)

This approach of using two ports is a good way to isolate the asynchronous code. One port is used for sending commands, configuration parameters, and outgoing data, and the other port is used for receiving incoming data and errors. How the service works internally is irrelevant to the CCR.

Blocking I/O

For long operations that involve blocking I/O requests, such as writing a large file, it is not desirable to tie up a thread. In this case you can create a separate dispatcher and submit a task to it. Alternatively, create a CLR thread and allocate it exclusively for use with the I/O operations. This removes it from the CCR scheduling.

Note that when you create a service (which is covered in Chapter 3), you can specify that you want it to have its own dispatcher, as described earlier in this chapter. This takes it out of contention for threads in the main DSS thread pool. You can do this using an attribute at the top of your service, as shown in the following example:

```
[ActivationSettings(ShareDispatcher=false, ExecutionUnitsPerDispatcher=6)]
[Contract(Contract.Identifier)]
public class CCRExamplesService : DsspServiceBase
{
    ...
}
```

Note that some blocking operations have Asynchronous Programming Model (APM) implementations that help with multi-threading. These are the classic begin/end operations that can be used with web requests (see the `IPCamera.cs` sample included in the MRDS distribution) or with file I/O. All that your code needs to do is create a port for signaling the completion of the asynchronous operation, execute the `Begin` method, and then post a message to this port in the `End` method (which runs asynchronously). Meanwhile, your CCR code can use a receiver to wait on the port.

This topic comes up again later in the section “Interoperation with Legacy Code.”

Throttling

A subtle way to enforce throttling is to set the number of threads to a small number on the dispatcher. Tasks will then pile up, waiting to execute, because not enough threads are available. Remember that you can apply policies to dispatcher queues that explicitly implement throttling strategies.

In Chapter 17, the code for the Hemisson Drive implements a throttling process for `SetDrivePower` requests by posting all requests to an internal port. There are several operations that you can use to change the drive power, and this “funneling” of the requests through a single port enables it to examine the queue and drop messages when required — a task that isn’t possible with the main interleave. This is necessary because serial communications with the Hemisson are very slow and must be done synchronously to avoid hanging the onboard monitor program. Recall that when you use a joystick to drive a robot, hundreds of requests can arrive in a short period of time.

Error Handling

You can, of course, still use structured exception handling with `try/catch/finally` statements. However, in a multi-threaded environment, it is possible for an error to occur in a different thread, and the error needs to be passed back to the original caller.

You have already seen examples of using a `Choice` to select either a valid response or a fault. This relies on the other service trapping any errors and generating a fault message.

If an exception occurs during the processing of a message in a DSS operation handler and you don't have a `try/catch`, then DSS automatically converts the `Exception` object to a `Fault` and sends it back as the response.

Note that DSS includes a set of methods for tracing. In particular, you can use the `LogError` method to send a message to the console output of the DSS node. This is explained in Chapter 3. It is always a good idea to put calls to `LogError` in places where the error might be catastrophic.

It is also possible to output information to the MRDS command window using the `Console.WriteLine` method. This is not a good approach, but it might be better than nothing at all. Be aware, however, that writing to the console this way is a relatively slow process; and you should not use it inside a tight loop or it will slow down your service. Microsoft discourages the use of `Console.WriteLine`.

Causalities

The CCR defines an error handling approach called *causalities*. These are based on the concept that no matter how you nest operations, and regardless of whether they are asynchronous or not, there is some point in the past that is the root cause of the current execution stream.

Causality contexts are passed from a sender to a receiver and propagate from there to any further receivers that are invoked. They can even handle multiple exceptions, such as might be generated from a `Join`, for example. In this case, the causality will contain multiple items.

Here is the example code from `CCRExamples`. In order to see it working, you must start the program without the debugger; otherwise, the debugger will trap the exception and the causality won't get a chance to handle it.

```
// Example of using a Causality
//
// NOTE: You must run this example without the Debugger!
// Otherwise the Debugger will trap the exception and you
// will not see the Causality working properly.
//
private void Causality()
{
    using (Dispatcher d = new Dispatcher())
    {
```

(continued)

Chapter 2: Concurrency and Coordination Runtime (CCR)

(continued)

```
using (DispatcherQueue taskQ = new DispatcherQueue("Causality Queue", d))
{
    Port<Exception> ep = new Port<Exception>();
    Port<int> p = new Port<int>();

    // Set up a causality for the current thread
    Dispatcher.AddCausality(new Causality("Test", ep));

    Console.WriteLine("Main thread: {0}",
Thread.CurrentThread.ManagedThreadId);

    // Set up a receiver that will generate an exception
    Arbiter.Activate(taskQ, Arbiter.Receive(false, p, CausalityExample));

    // Post an item and the causality goes with it
    p.Post(2);
    Wait(500);

    Exception e;
    while (ep.Test(out e))
        Console.WriteLine("Exception: " + e.Message);
    }
}

// Receiver to execute with causality enabled
private void CausalityExample(int num)
{
    Console.WriteLine("CausalityExample thread: {0}",
Thread.CurrentThread.ManagedThreadId);

    Port<int> p = new Port<int>();
    // Set up another receiver
    Arbiter.Activate(Environment.TaskQueue,
    Arbiter.Receive(false, p,
    // And now an anonymous delegate
    delegate(int n)
    {
        Console.WriteLine("Anonymous method thread: {0}",
Thread.CurrentThread.ManagedThreadId);
        // Can you spot the deliberate error?
        int i = 0; n = n / i;
    }
    ));
    // Post a message to activate the receiver
    p.Post(num);
}
```

The output from the preceding example might look as follows:

```
Main thread: 8
CausalityExample thread: 15
Anonymous method thread: 9
Exception: Attempted to divide by zero.
```

Notice that three different threads are involved here:

- ❑ The main thread is the one that displays the exception message because it does not yield (it just sleeps).
- ❑ The second thread is the receiver called `CausalityExample`, and it is deliberately run in a different dispatcher to illustrate the point.
- ❑ The third thread is from the default dispatcher and is a delegate receiver inside `CausalityExample`.

One other point to note about the preceding code is the use of a `using` statement to wrap all of the code in the first routine that uses the new dispatcher and dispatcher queue. This ensures that these are disposed of as soon as the block of code finishes executing, rather than waiting for the garbage collector. The other examples in `CCRExamples` didn't bother with this, although they probably should have done so.

Success/Failure Ports

The next example shows how to use a `SuccessFailurePort`, which is a `PortSet` consisting of a `SuccessResult` and an `Exception`. In this case, the handler is the `GetHandler` from the `Lynx6Arm` service in Chapter 15. The overall approach here is to call a method that returns a port and then branch based on the result.

```
/// <summary>
/// Get Handler
/// </summary>
/// <param name="get"></param>
/// <returns></returns>
/// <remarks>Get the state for the service</remarks>
private IEnumerator<ITask> GetHandler(Get get)
{
    yield return Arbiter.Choice(
        UpdateState(),
        delegate(SuccessResult success)
        {
            get.ResponsePort.Post(_state);
        },
        delegate(Exception ex)
        {
            get.ResponsePort.Post(Fault.FromException(ex));
        }
    );
}
```

The `GetHandler` sets up a `Choice` that calls `UpdateState`, which returns a `SuccessFailurePort`. The `Choice` arbiter listens for a message, which will be either a `SuccessResult` or an `Exception`. If a `SuccessResult` is returned, then the state is sent back on the `ResponsePort`. The state is global and is updated by `UpdateState`.

Because `Get` operations are intended to be accessible via the Web, the handler must return either the current state or a SOAP `Fault`. Therefore, if an exception occurs, the `GetHandler` has to convert it to a `Fault` before posting it back to the `ResponsePort` specified in the `Get` request. This can be done using

Chapter 2: Concurrency and Coordination Runtime (CCR)

the `FromException` static method on the `Fault` class. Note that you need to include a `using` statement for `W3C.Soap` in order to use `Fault`.

The `UpdateState` method creates a new `SuccessResultPort` and then posts either a `SuccessResult` or an `Exception` to it based on the result of a `Query` command sent to the SSC-32 servo controller. In this case, the code has to convert from a `SOAP Fault` back to an `Exception` in order to return a failure status. Note that `UpdateState` actually returns a port with a message already sitting in it. This enables a `Choice` to be used in the `GetHandler`.

```
// Synchronize the state by reading the joint positions from the controller
private SuccessFailurePort UpdateState()
{
    SuccessFailurePort resultPort = new SuccessFailurePort();

    //Create new query pulse width command
    int[] channels = new int[Lynx6ArmState.NUM_JOINTS] { 0, 1, 2, 3, 4, 5 };
    ssc32.SSC32QueryPulseWidth queryCommand = new ssc32.SSC32QueryPulseWidth();
    queryCommand.Channels = channels;

    ssc32.SendSSC32Command command = new ssc32.SendSSC32Command(queryCommand);
    _ssc32Port.Post(command);

    //Update the arm state based on the query response
    Activate(Arbiter.Choice(command.ResponsePort,
        delegate(ssc32.SSC32ResponseType response)
        {
            ssc32.SSC32PulseWidthResponse queryResponse =
(ssc32.SSC32PulseWidthResponse)response;
            for (int i = 0; i < _state.Joints.Count; i++)
            {
                int jointAngle = ServoAngleToJointAngle(i,
PulseWidthToAngle(queryResponse.PulseWidths[i]));
                _state.Joints[i].State.Angular.DriveTargetOrientation =
AngleToOrientationQuaternion(jointAngle);
                // Remember the angle too - much easier to use!
                _state.Angles[i] = jointAngle;
            }
            _state.GripperAngle =
(int)Math.Round(_state.Angles[(int)JointNumbers.Gripper]);

            resultPort.Post(new SuccessResult());
        },
        delegate(Fault fault)
        {
            resultPort.Post(new Exception(fault.Reason[0].ToString()));
        }
    ));

    return resultPort;
}
```

Interoperation with Legacy Code

In terms of the CCR, Windows Forms (WinForms) constitute legacy code. WinForms use a STA model that does not work well under the CCR. Therefore, creating and using WinForms requires special treatment. See Chapter 3 for more information.

Using GDI graphics primitives suffers from problems similar to WinForms. This includes manipulating bitmaps. You need to be careful to ensure that you don't end up with an access violation.

The Webcam service also needs to run in STA mode in order to use COM interop with older-style webcam drivers. For the STA model, you should create a dispatcher with a single thread.

The MTA (Multi-Threaded Apartment) model is a little different. If you want, or need, a fixed set of threads, then create a dispatcher with the appropriate number of threads. This will isolate these threads from the threads in the default DSS dispatcher.

If you have a requirement to use older code that uses the APM with `Begin/End` operations, such as asynchronous file I/O, then you can wrap these operations in classes that use CCR ports. For further information on this topic, search the discussion forum. There is also a good MSDN article called "Concurrent Affairs" by Jeffrey Richter on the MSDN Magazine website. It includes code for an APM wrapper. The URL is <http://msdn.microsoft.com/msdnmag/issues/06/09/ConcurrentAffairs/default.aspx>.

Traps for New Players

Newcomers to MRDS often make a few common mistakes. This section briefly describes these and how to avoid them:

- ❑ In conventional code — i.e., not in an iterator — if you forget to use `Activate` on a `Receive`, `Choice`, and so on, then nothing will happen. The code will compile, but it will not do what you expect.
- ❑ If you call an iterator method directly, then nothing will happen. You must use `SpawnIterator` or some other method that activates it as a task. Again, the code will compile, but you will spend some time tearing your hair out wondering why it doesn't do anything and breakpoints inside the method never fire.
- ❑ Iterator syntax is incompatible with `out` and `ref` parameters, i.e., you cannot return anything. However, you can pass parameters into an iterator. The only workaround for this is to pass in a `Port<type>` and then post back an instance of `type` (which is some appropriate data type that you choose).

One way to call an iterator with a parameter is as follows:

```
yield return CcrHelpers.FromIteratorHandler(3, TestMethod);
```

Chapter 2: Concurrency and Coordination Runtime (CCR)

where the CCR helper function is defined as follows:

```
static class CcrHelpers
{
    public static ITask FromIteratorHandler(T0 t0, IterativeHandler handler)
    {
        yield return Arbiter.ExecuteToCompletion(
            base.TaskQueue,
            new IterativeTask<T0>(t0, handler));
    }
}
```

The `SpawnIterator` method has three overloads that take one, two, or three parameters and can be used as shown in the following example:

```
SpawnIterator<int, bool>(42, true, handler);
```

- ❑ `Arbiter.ExecuteToCompletion` activates the task. However, you still need to use `yield return` or the code does not wait for completion. This is a little confusing.
- ❑ Although this next tip really belongs in the next chapter, it is still relevant here. A service has one `interleave`, `MainPortInterleavePort`, which is created by DSS. If you set up other independent receivers, then they are not afforded the protection of the Exclusive Receiver Group and will run concurrently. This can result in very strange behavior because the state of the service might be updated simultaneously by two threads. The solution is to use the function `MainPortInterleave.CombineWith`.

Similarly, when you use `SpawnIterator` or `Activate`, these tasks run in parallel with tasks from the main `interleave`.

Summary

The CCR is a very efficient and lightweight runtime. It allows multi-threaded programming without many of the complexities of a traditional approach. However, for many people it represents a different programming paradigm and it takes a little bit of getting used to. This chapter has introduced you to a variety of different methods in the CCR.

Although a lot of the CCR has been covered, many other APIs have not been discussed. In addition to reading the documentation, you can also use the Microsoft Robotics Developers Studio discussion forum as a resource. Use the Search function first. If you can't find an answer to your question, then post a message. The forum is at <http://forums.microsoft.com/MSDN/default.aspx?ForumGroupID=383&SiteID=1>.

Chapter 3 discusses Decentralized Software Services (DSS). Using DSS is essential for writing MRDS services.