Introduction to Game Programming and Robotics

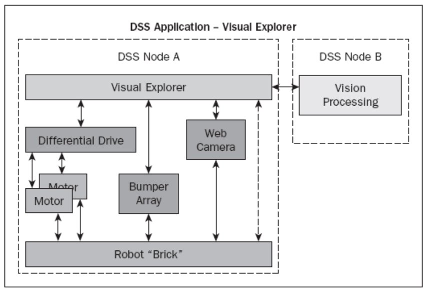
Unit # 6

DSS - Introduction

- The Decentralized Software Services (DSS) are responsible for controlling the basic functions of robotics applications.
- They are responsible for starting and stopping services and managing the flow of messages between services via *service forwarder ports*.
- There is a DSS base class from which all services are derived, and this draws heavily on the features of the CCR.

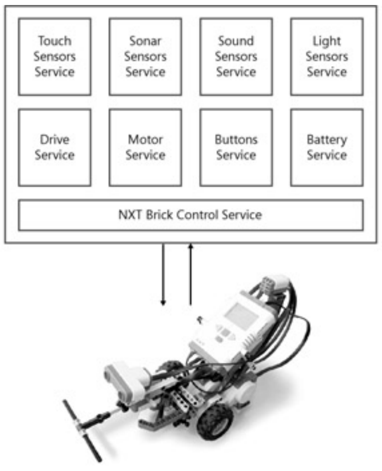
Distributed Computing

- Services can be used to represent anything including, but not limited to:
  - Hardware components such as sensors and actuators
  - Software components such as User Interface (UI), storage, directory services, etc.
- Services are inherently network enabled and can communicate with each other in a uniform manner. This works regardless of whether they are executed within the same DSS node or across the network.



Orchestration

- The basic building block in MRDS is a service .
- Services can be combined (or composed) as *partners* to create applications .
- This process is referred to as *orchestration*.
- A robotics application consists of multiple services that work together to achieve a common task—operating the robot.

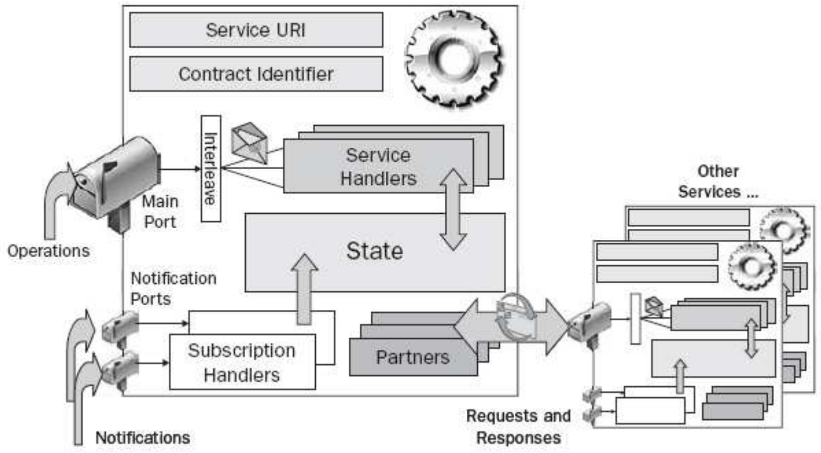


Composition of a Service

- Contract
  - This defines the messages you can send to a service, as well as a globally unique reference, called the *contract identifier* , which identifies the service and is expressed in the form of a *URI* (Universal Resource Identifier).
- State
  - Information that the service maintains to control its own operation
- Behavior
  - The set of *operations* that the service can perform and that are implemented by *handlers*
- Execution Context
  - The *partnerships* that the service has with other services, and its initial state

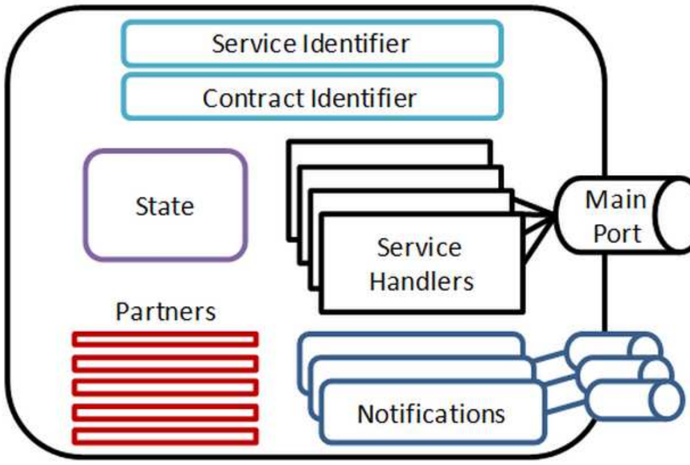
Sajjad Haider Fall 2009 5

Composition of a Service (Cont'd)



Sajjad Haider Fall 2009 6

Composition of a Service (Cont'd)



Creating a New Service

- To create a new service, open Visual Studio, choose File, New, and then click Project.
- In the New Project dialog box, expand the node for the language you prefer, select the Robotics node, and then select the Simple DSS Service (1.5) project template.
- In the Name text box, type a name for the new service, enter a location for the code files, and then click OK. This will create the solution and files you need to build a simple DSS service.

Create a New Service

- Four files created by Visual Studio
  - AssemblyInfo.cs: You don’t need to concerned about it for the time being.
  - ServiceA.cs: Main source file. Most of the work of creating a new service is done in this file.
  - ServiceA.manifest.xml: This is the manifest that is used by DSS to load the service.
  - ServiceAType.cs: This contains a set of classes, also called *types*, that are used by the service and other services that which to communicate with it.

Understanding the Project Files

- DssService1.cs is the implementation class, and this is where you will place the code that reads data from sensors and sends commands to your robot.
- DssService1-Types.cs is the contract class, and this is where you will return information about the service such as the *state*. The *contract* class will also handle any requests to drop or create the service.

Contract Identifier

- When a service instance is created within a DSS Node, it is dynamically assigned a Universal Resource Identifier (URI), by the constructor service.
- Every service must have a class called *Contract*, and it must contain a field called *Identifier* .
- The *Contract* class is a mandatory part of the contract, and it must contain a string called *Identifier*.

```
public sealed class Contract
   {

      /// <summary>
      /// The Dss Service contract
      /// </summary>
      public const String Identifier =
     "http://schemas.tempuri.org/2009/10/dssservice4.html";
   }
```

State

- The service state is a representation of a service at any given point in time. One way to think of the service state is as a document that describes the current content of a service.
  - The state of a service representing a motor may consist of rotations per minute, temperature, oil pressure, and fuel consumption.
  - If you have a service that returns data from a sensor, then the state for that service would be the data read from the sensor at the time you requested the state.
  - A service representing a keyboard may contain information about which keys have been pressed.
- Any information that is to be retrieved, modified, or monitored as part of a DSS service must be expressed as part of the service state.

State Class

- All services have a state class.
- Adding fields to the state class is one of the standard steps in creating a new service.

```
/// <summary>
/// The DssService4 State
/// </summary>
[DataContract]
public class DssService4State
{
}
```

Main Port

- The main port is a CCR Port where messages from other services arrive.
- It is also known as the operations port. Because service implementations do not link against each other directly, a service can only talk to another service by sending a message to its main port.
- The messages accepted on the main port are defined by the type of the port. In particular, there must be at least one *PortSet* that is public and contains ports for all of the available message types.

Main Port (Cont'd)

- A service's main operations port must have the [ServicePort] attribute. Note that even though it is called a port, it is really a PortSet that lists all of the operations supported by the service.

```
/// <summary>
/// DssService4 Main Operations Port
/// </summary>
[ServicePort()]
public class DssService4Operations :
 PortSet<DsspDefaultLookup, DsspDefaultDrop, Get>
{
}
```

Main Port (Cont'd)

- The default *PortSet* when you create a new service only contains *DsspDefaultLookup*, *DsspDefaultDrop*, and *Get* operations.
- There are no definitions for *DsspDefaultLookup* and *DsspDefaultDrop* because these operations are implicitly handled by the DsspServiceBase class.

Get

- *Get* is just one of several DSSP operations allowed.
- It will return state in the format of a SOAP message as required by DSSP.

Get Operation

```
/// <summary>
   /// DssService4 Get Operation
   /// </summary>
   public class Get : Get<GetRequestType, PortSet<DssService4State, Fault>>
   {

      /// <summary>
      /// DssService4 Get Operation
      /// </summary>
      public Get()
      {
      }

      /// <summary>
      /// DssService4 Get Operation
      /// </summary>
      public Get(Microsoft.Dss.ServiceModel.Dssp.GetRequestType body) :
            base(body)
      {
      }

      /// <summary>
      /// DssService4 Get Operation
      /// </summary>
      public Get(Microsoft.Dss.ServiceModel.Dssp.GetRequestType body, Microsoft.Ccr.Core.PortSet<DssService4State,W3C.Soap.Fault> responsePort) :
            base(body, responsePort)
      {
      }
   }
```

Service.cs Class

- Every service must have an instance of its service state, even if the state is empty, and a main operations port.
- By convention, the service state is called _state and the operations port is called _mainPort, but you can call them anything you like.

```
/// <summary>
/// _state
/// </summary>
private DssService4State _state = new DssService4State();

/// <summary>
/// _main Port
/// </summary>
[ServicePort("/dssservice4", AllowMultipleInstances=false)]
private DssService4Operations _mainPort = new DssService4Operations();
```

Messages Handling

```
[ServicePort("/dssservice4", AllowMultipleInstances=false)]
 private DssService4Operations _mainPort = new
  DssService4Operations();
```

- This code indicates that the main port does not allow multiple instances and that the service will be located in a subdirectory named dssservice4.
- Given this information, you can determine the URL of the service, *http://localhost:50000/dssservice4*.

Constructor

- The constructor of the class is always empty.

```
/// <summary>
/// Default Service Constructor
/// </summary>
public
DssService4Service(DsspServiceCreationPort
creationPort) :  base(creationPort)
{
}
```

Start Method

- All the services have a Start method which is called during service creation so that the service can initialize itself.

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

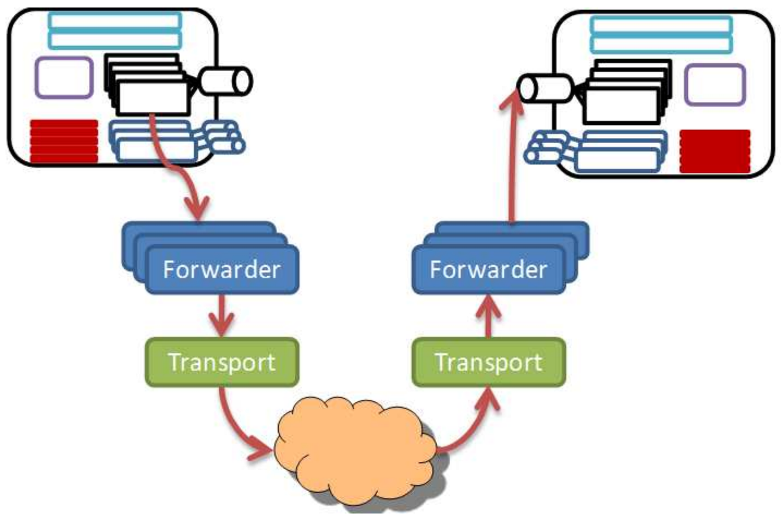
Start Method (Cont'd)

- Calling **base.Start()** does three things for the service.
  - Calls **ActivateDsspOperationHandlers** which causes **DsspServiceBase** to attach handlers to each message supported on the main service port. (This relies on **ServiceHandler** attributes and method signatures.)
  - Calls **DirectoryInsert** to insert the service record for this service into the directory. The directory is itself a service and this method sends an **Insert** message to that service.
  - Calls **LogInfo** to send an **Insert** message to the **/console/output** service (**http://localhost:50000/console/output**). The category of the message is **Console**, which causes the message to be printed to the command console. The URI of the service is automatically appended to the output.

Service Handlers

- For each of the DSSP operations defined on the main port, service handlers need to be registered to handle incoming messages arriving on that port.
- Service handlers can be registered declaratively using the **ServiceHandler** attribute.
- Messages are sent through a **service forwarder** which is a local Concurrency and Coordination Runtime Port representing the main port of the remote service.
- When a message is sent through the service forwarder, it gets forwarded down through the runtime until it reaches a transport.
- This transport will route the message, possibly through the network, to the transport of the other service.
- Here the message will get forwarded back up through the runtime until it reaches the main port of the receiving service.

Service Handlers (Cont'd)



Service Handlers (Cont'd)

- A service handler is responsible for handling incoming messages on a port. A service can have more than one service handler, and each one will pertain to a different DSSP, HTTP, or custom operation.
- The Visual Studio template, which is used to create a new DSS service, provides only one service handler.

```
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public virtual IEnumerator<ITask> GetHandler(Get get)
{
    get.ResponsePort.Post(_state);
    yield break;
}
```

Service Handlers (Cont'd)

- The *GetHandler* does not do much. It simply posts the state as a SOAP message to the response port.
- The code in this handler could contain additional code that performs computations or sends messages to other services.
- It is annotated with the [*ServiceHandler*] attribute, which specifies that it belongs to the Concurrent group.
- This handler is allowed to execute simultaneously with other Concurrent handlers, but it is not allowed to execute while an Exclusive handler is running.

Adding State Variable

- If you want the DssService project to do something simple, then you can have it return the phrase "Hello World."
- In this case, the phrase "Hello World" is the service's state, and this phrase is returned when someone accesses the service.
- To accomplish this, you would need to add the following code to the DssService1Types.cs file (inside the public class definition for the *DssService1State* class):

Adding State Variable (Cont’d)

```
/// <summary>
/// The DssService3 State
/// </summary>
[DataContract]
public class DssService3State
{
   private string _outputmsg = "Hello World" ;

   [DataMember]
   public string OutputMsg
   {
      get { return _outputmsg; }
      set { _outputmsg = value; }
   }
}
```

Adding State Variable (Cont’d)

- Add an integer variable _number.

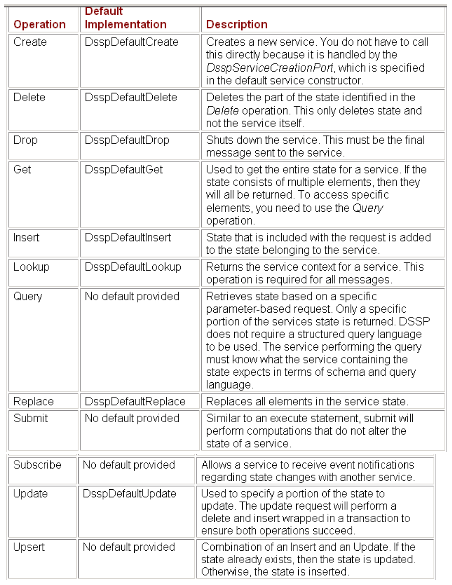
```
private int _number = 1;
public int Number
     {
        get { return _number; }
        set { _number = value; }
     }
```

- Also add the following code in the Start() function:

```
for (int i = 0; i < 10; i++)
           _state.Number = i;
```

- Note that we haven’t added the [DataMember] attribute yet. Run the program and open the service in the web browser. Do you find this attribute there?
- Now add [DataMember] and see the difference.

DSSP Operations



Operation	Default Implementation	Description
Create	DsspDefaultCreate	Creates a new service. You do not have to call this directly because it is handled by the *DsspServiceCreationPort*, which is specified in the default service constructor.
Delete	DsspDefaultDelete	Deletes the part of the state identified in the *Delete* operation. This only deletes state and not the service itself.
Drop	DsspDefaultDrop	Shuts down the service. This must be the final message sent to the service.
Get	DsspDefaultGet	Used to get the entire state for a service. If the state consists of multiple elements, then they will all be returned. To access specific elements, you need to use the *Query* operation.
Insert	DsspDefaultInsert	State that is included with the request is added to the state belonging to the service.
Lookup	DsspDefaultLookup	Returns the service context for a service. This operation is required for all messages.
Query	No default provided	Retrieves state based on a specific parameter-based request. Only a specific portion of the services state is returned. DSSP does not require a structured query language to be used. The service performing the query must know what the service containing the state expects in terms of schema and query language.
Replace	DsspDefaultReplace	Replaces all elements in the service state.
Submit	No default provided	Similar to an execute statement, submit will perform computations that do not alter the state of a service.
Subscribe	No default provided	Allows a service to receive event notifications regarding state changes with another service.
Update	DsspDefaultUpdate	Used to specify a portion of the state to update. The update request will perform a delete and insert wrapped in a transaction to ensure both operations succeed.
Upsert	No default provided	Combination of an Insert and an Update. If the state already exists, then the state is updated. Otherwise, the state is inserted.

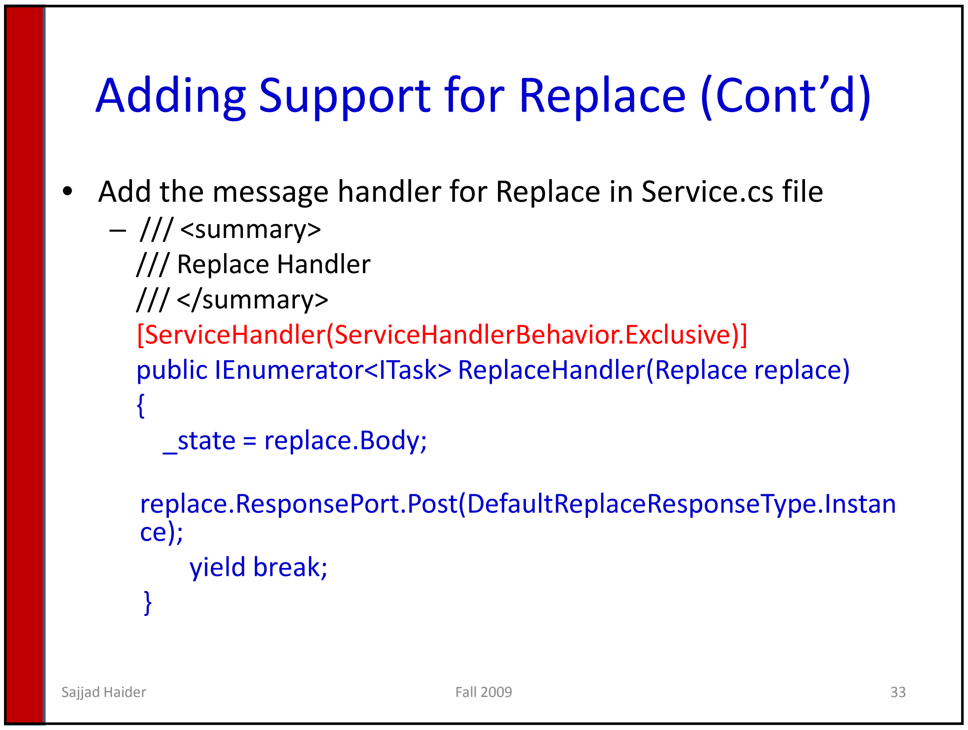
Adding Support for Replace

- Add the **Replace** message to the list of messages supported by the service's port.
  - [ServicePort]

    public class ServiceTutorial1Operations : PortSet<DsspDefaultLookup, DsspDefaultDrop, Get, Replace> { }
- Add the following
  - public class Replace : Replace<ServiceTutorial1State, PortSet<DefaultReplaceResponseType, Fault>>

    {

    }

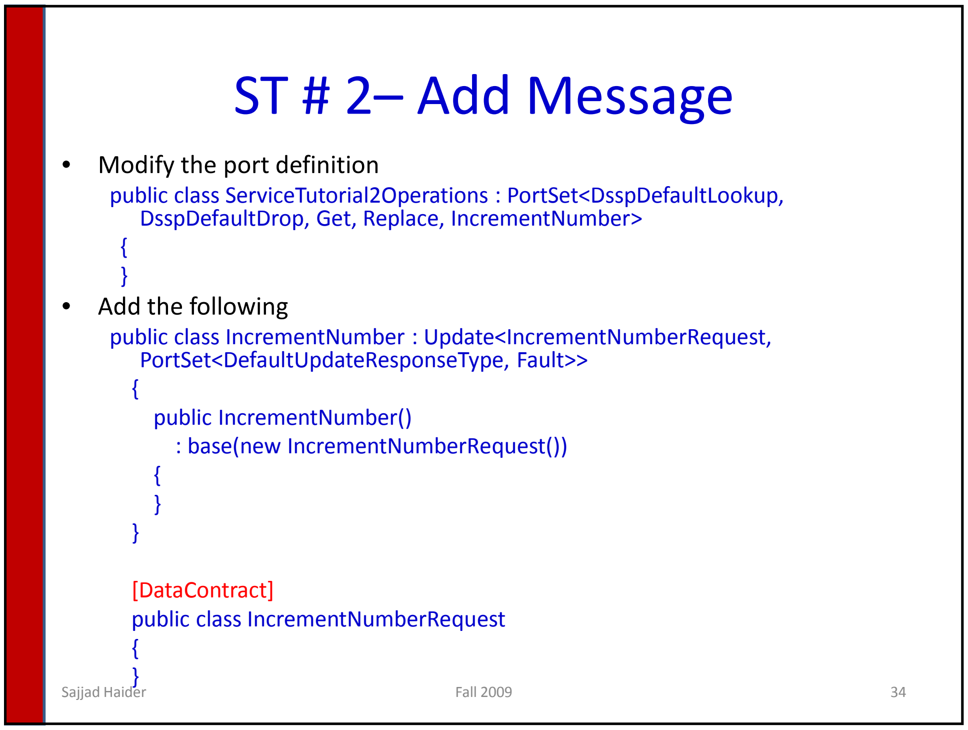


Adding Support for Replace (Cont'd)

- Add the message handler for Replace in Service.cs file

  -
    ```
    /// <summary>
    /// Replace Handler
    /// </summary>
    [ServiceHandler(ServiceHandlerBehavior.Exclusive)]
    public IEnumerator<ITask> ReplaceHandler(Replace replace)
    {
      _state = replace.Body;

    replace.ResponsePort.Post(DefaultReplaceResponseType.Instan
    ce);
          yield break;
     }
    ```



ST # 2– Add Message

- Modify the port definition
```
public class ServiceTutorial2Operations : PortSet<DsspDefaultLookup,
    DsspDefaultDrop, Get, Replace, IncrementNumber>
 {
 }
```
- Add the following
```
public class IncrementNumber : Update<IncrementNumberRequest,
    PortSet<DefaultUpdateResponseType, Fault>>
   {
     public IncrementNumber()
        : base(new IncrementNumberRequest())
     {
     }
   }

   [DataContract]
   public class IncrementNumberRequest
   {
   }
```

ST # 2 – Service Handler

- Add the message handler in service.cs file

```
[ServiceHandler(ServiceHandlerBehavior.Exclusive)]
public IEnumerator<ITask>
   IncrementNumberHandler(IncrementNumber
   incrementNumber)
{
      _state.Number++;
      LogInfo("Tick: " + _state.Number);

   incrementNumber.ResponsePort.Post(DefaultUpdateRespo
   nseType.Instance);
      yield break;
}
```

ST # 2 – Service Handler (Cont’d)

- The service increments the **Number** property approximately once per second.
- To do this, it needs some kind of timer.
- The first thing you have to do is declare a port to which a message is sent each time the timer fires.
- The following line declares a port to which a **DateTime** can be posted. Add this member field in the service class after the **_mainPort** declaration.

ST # 2 – Add New Port

- Add this member field in the service class after the **_mainPort**declaration.
  - private Port<DateTime> _timerPort = new Port<DateTime>();
- Add two lines to the **Start** method:
  - Post a **DateTime** to the port you have just declared. The value of **DateTime** posted is unimportant to the execution of the service although it can be useful for debugging.
  - Activate a handler for the port **_timerPort**.

```
protected override void Start()
{
base.Start(); // Kick off the timer (with no delay) and start a receiver
   for it
_timerPort.Post(DateTime.Now);
Activate(Arbiter.Receive(true, _timerPort, TimerHandler));
}
```

ST # 2 – New Handler

```
void TimerHandler(DateTime signal)
    {
      _mainPort.Post(new IncrementNumber());

      Activate(
        Arbiter.Receive(false, TimeoutPort(1000),
          delegate(DateTime time)
          {
            _timerPort.Post(time);
          }
        )
      );
    }
```

ST # 2 – New Handler (Cont’d)

- **TimerHandler** is called when a message arrives on **_timerPort**. It does two things:
  - It posts an **IncrementNumber** message to the main service port. This causes the execution of **IncrementNumberHandler**.
  - It activates on a 1000 millisecond timeout interval. Since we have not declared a method as the handler for this receiver, an anonymous delegate (a new feature in .NET 2.0) is used. This allows us to write the handler inline in the call to **Arbiter.Receive**. In that anonymous delegate, we post the **DateTime** value to the port,**_timerPort**. Note that this receiver is not persistent--the first parameter is **false**. This is because the port created by the **TimeoutPort()** method receives one message after the specified interval (in this case 1000 milliseconds) expires.

ST # 2 – New Handler (Cont’d)

- When the **Activate()** method is called, it adds the task defined by calling **Arbiter.Receive** to the list of active tasks for this service.
- The operations defined within the **Activate** call will be scheduled independently of the current thread.
- In this case the **TimerHandler** method will return immediately after the call to **Activate**; it does not wait until the timer is fired.
- The **Arbiter.Receive()** method defines a one-time receiver that will take the message sent when the timer interval specified in the call to **TimeoutPort** expires (in this case after 1000 milliseconds) and passes it as the parameter to the anonymous delegate specified as the third parameter to **Arbiter.Receive()**.

Robotics Tutorial # 1

- This tutorial teaches you how to use a basic service that reads the output of a contact (touch) sensor and displays a message in the Console window.

Tutorial # 1: Step 1

- Create a new project.
- Add reference to RoboticsCommon.Proxy.
- At the top of service.cs file, add
  - using bumper = Microsoft.Robotics.Services.ContactSensor.Proxy;
- Now create a partnership between your service and the bumper service.

Tutorial # 1: Step 2

- To communicate with the bumper service, we set up a port of **ContactSensorArrayOperations** and identify this as a partner by using the **Partner** attribute.
- Add the following code to your service after the line that defines **_mainPort**
  - [Partner("bumper", Contract = bumper.Contract.Identifier, CreationPolicy = PartnerCreationPolicy.UseExisting)]

    private bumper.ContactSensorArrayOperations _bumperPort = new bumper.ContactSensorArrayOperations();

Tutorial # 1: Step 2 (Cont'd)

- The simplest way to bind the service partner to your hardware is to start an additional manifest which contains the service contract(s) for your hardware.
- Modify Project Properties → Debug
  - /p:50000 /t:50001 /m:"samples/MyTutorial1/MyTutorial1.manifest.xml" /m:"samples/config/LEGO.NXT.MotorTouchSensor.manifest.xml“

Tutorial # 1: Step 3

- Modify the **Start()** method to subscribe to the bumper service and begin listening for contact sensor notifications.
- Add a call to the **SubscribeToBumpers** method to subscribe to the bumper service.
  - SubscribeToBumpers();

Tutorial # 1: Step 4

- Write the subscription.

```
/// <summary>
/// Subscribe to the Bumpers service
/// </summary>
void SubscribeToBumpers()
{ // Create the bumper notification port.
  bumper.ContactSensorArrayOperations bumperNotificationPort =
  new bumper.ContactSensorArrayOperations();
  // Subscribe to the bumper service, receive notifications on the
  bumperNotificationPort.
  _bumperPort.Subscribe(bumperNotificationPort);
  // Start listening for updates from the bumper service.
  Activate( Arbiter.Receive<bumper.Update> (true,
  bumperNotificationPort, BumperHandler));
}
```

Tutorial # 1: Step 4 (Cont'd)

- The first task of the **SubscribeToBumpers()** method is to create a notification port on which to receive notifications from the bumper service. Create a notification port by creating an instance of **ContactSensorArrayOperations**.
- Subscribe to the **_bumperPort** port, and specify that notifications be sent to the **bumperNotificationPort**.
- Use the **Activate()** method to set up the handler to receive notifications from the bumper.

Tutorial # 1: Step 5

- Add the BumperHandler().

```
/// <summary>
/// Handle Bumper Notifications
/// </summary>
/// <param name="notification">Update
    notification</param>
private void BumperHandler(bumper.Update notification)
{
    if (notification.Body.Pressed)
        LogInfo(LogGroups.Console, "Ouch - the bumper
        was pressed.");
}
```