Basic DOF Operations

C Programmer's Guide

Version 7.0

Table of Contents

Chapter 1: Introduction 1

How to Read This Guide 1

The Time-Based Alarm Interface 1

Description of Interface Items 2

Item Identifiers 2

Interface Code 3

Chapter 2: Basic Setup 5

Initializing the Library 5

Obtaining DOFSystem Instances 5

Creating DOFObjectID Instances 6

Creating DOFObject Instances 7

Making a DOFObject a Provider 7

Implementing the Provider Interface 7

Initiating a Provide Operation 8

Default Provider Behavior 10

Chapter 3: Synchronous Operations 12

Getting Properties Synchronously 12

Provider 12

Requestor 14

Setting Properties Synchronously 15

Provider 15

Requestor 16

Invoking Methods Synchronously 17

Provider 18

Requestor 19

Chapter 4: Exception Handling 21

DOF Exception Types 21

Returning Exceptions in Provider Applications 21

Provider Exceptions 21

Error Exceptions 23

Catching Exceptions in Requestor Applications 24

Chapter 5: Interest and Query Operations 26
Interest Operations **26**
Relationship Between Interest and Provide Operations 26
Interest Levels 27
Query Operations **27**
Discovering a Single Provider **27**
Expressing Interest 27
The WaitProvider Query 29
Customized Query Operations **29**
Building a Query 30
Implementing a Query Listener 32
Discovering Multiple Providers **33**
Expressing Interest 33
Initiating a Query Operation 34

Chapter 6: Single-Transaction Asynchronous Operations 35
Getting Properties Asynchronously **35**
Get Operation Listener 36
Calling DOFObject_BeginGet 37
Setting Properties Asynchronously **37**
Set Operation Listener 38
Calling DOFObject_BeginSet 38
Invoking Methods Asynchronously **39**
Invoke Operation Listener 39
Calling beginInvoke DOFObject_BeginInvoke 40

Chapter 7: Ongoing Asynchronous Operations 42
Subscribing to a Property **42**
Providers 43
Subscribe Operation Listener 43
Calling DOFObject_BeginSubscribe 44
Registering for an Event **45**
Providers 46
Register Operation Listener 46
Calling DOFObject_BeginRegister **47**

Chapter 8: Activate-Level Interest 49
Implementing DOFSystemActivateInterestListener **49**
Writing the activate Method **50**
Writing the cancelActivate Method **51**

Appendix A: Formatting OIDs 52
GUID Class OIDs 52
Domain Class OIDs 52
Email Class OIDs 53
Formatting OIDs for DOFObjectID Create Methods 53
Summary 54

Chapter 1: Introduction

The *Basic DOF Operations C Programmer's Guide* contains the basic knowledge that applications programmers need to get started with the DOF Object Access Library (OAL). The DOF OAL is currently available in Java, C, and C#.

DOF is an object-oriented technology at its core, specifically modeled on Java language conventions. If you are unfamiliar with object-oriented programming in general or with specific Java definitions of terms such as *field* and *interface*, you may want to familiarize yourself with these before working with any of the OALs, *even if you are not working in Java*. A basic understanding of Java will help you to better understand DOF.

How to Read This Guide

We recommend you read the entire document, making sure you understand each chapter before going on to the next. Classes and functionality that are used in multiple operations are introduced in the first operation where they occur. This allows them to be introduced in context, making them easier to understand; however, the information is not repeated when they are used a second time. After you've gone through the guide and learned the basic concepts, the document will become useful as a reference tool.

Before reading this guide, we recommend you read the *Interface Design Concepts Training Guide,* which is found in the DOF Essentials SDK.

This guide can also be used with the operations training in the DOF Essentials SDK. The training materials include the source code shown in this guide and may be copied into your own development environment and used for practice. Although samples shown in this guide are based on the training code, the actual code may vary slightly.

Note: The functionality required to make the training code compile and run is often omitted from the samples so you can focus on the lines of code that are most relevant to performing the DOF operations.

The Time-Based Alarm Interface

To illustrate DOF operations, the code samples in this guide are based on an interface for a time-based alarm system. The following ideas were behind the basic design for this system:

- Many requestors would be somewhat limited devices that include only an alarm (lacking even their own clocks).
- A few requestors would be management applications that can set the alarm time for other devices.

- Providers would be applications capable of storing the alarm time for multiple devices and sending a notification to those devices when their alarms should be triggered.

To enable you to work with the DOF interface for this system, the system designer who created the interface would give you three types of information:

1. A description of the interface items
2. The item identifiers for all of the interface items
3. Code for the interface

The section that follow provide descriptions of each information type.

Description of Interface Items

The interface defines the following five items of functionality:

1. **AlarmActive.** This is a writable property that stores a boolean for whether the alarm is set or not. Providers must store the boolean value. Providers must enable requestors to get or subscribe to the value. They must also enable requestors to set the value. A *true* value means the alarm is on.
2. **AlarmTimeValue.** This is a read-only property that stores a date-time value. Providers must store the value and enable requestors to get the value. Providers do not need to implement set operations for this value since the property is read-only.
3. **SetNewTime.** This is a method that allows requestors to set the alarm time. Requestors will send a new alarm time as an input to the method. Providers must change the AlarmTimeValue to the input value in response to this method. In addition, the method must return the boolean for the active value.
4. **AlarmTriggered.** Providers must program their applications to trigger this event when the current system time is equal to the AlarmTimeValue and the AlarmActive property is set to true. Providers must also enable requestors to register to the event. The event has no outputs.
5. **BadTimeValue.** Providers must program their applications to throw this exception when a requestor attempts to set an invalid time value in the SetNewTime method.

Item Identifiers

Every item in an interface is assigned an identifier. These item identifiers must be unique within an interface and may be as large as 2^{16}-1; however, because most interfaces have only a few items, item identifiers will typically be small. Item identifiers for the Time-Based Alarm interface are shown in the following table.

Item	Identifier
AlarmActive property	1
AlarmTimeValue property	2

Item	Identifier
SetNewTime method	3
AlarmTriggered event	4
BadTimeValue exception	5

Interface Code

In this section, we'll examine the interface code. While some interface code can be more basic than shown in the example below, header file initialization and shutdown routines were included as a convenience for accessing needed information from the interface.

Sample Code

The following shows the C file created for the Time-Based Alarm interface:

```
#include "TimeBasedAlarmInterface.h"

DOFInterface TimeBasedAlarmInterface_DEF;
DOFInterfaceID TimeBasedAlarmInterface_IID;

uint8 TimeBasedAlarmInterface_IDBytes[] = { 0x07, 0x01, 0x00, 0x00, 0x34 };
uint8 TimeBasedAlarmInterface_Bytes[] = {
   0x02,
   0x40,
   0x42,
   0x02,
   0x01, 0x03, 0x01,
   0x02, 0x01, 0x00,
   0x01,
   0x04, 0x00,
   0x01,
   0x03, 0x01, 0x00, 0x01, 0x01,
   0x01,
   0x05, 0x00,
};

void TimeBasedAlarmInterface_Init(void){
   TimeBasedAlarmInterface_IID = DOFInterfaceID_Create_Bytes
            (sizeof(TimeBasedAlarmInterface_IDBytes),
            TimeBasedAlarmInterface_IDBytes);
   TimeBasedAlarmInterface_DEF = DOFInterface_Create(TimeBasedAlarmInterface_IID,
            sizeof(TimeBasedAlarmInterface_Bytes),
            TimeBasedAlarmInterface_Bytes);
}

void TimeBasedAlarmInterface_Shutdown(void){
   DOFInterface_Destroy(TimeBasedAlarmInterface_DEF);
   DOFInterfaceID_Destroy(TimeBasedAlarmInterface_IID);
}
```

The following shows the header file:

```
#include <dof/oal.h>

#define TimeBasedAlarm_AlarmActiveID 1
```

```
#define TimeBasedAlarm_AlarmTimeValueID 2
#define TimeBasedAlarm_SetNewTimeID 3
#define TimeBasedAlarm_AlarmTriggeredID 4
#define TimeBasedAlarm_BadTimeValueID 5

extern DOFInterface TimeBasedAlarmInterface_DEF;
extern DOFInterfaceID TimeBasedAlarmInterface_IID;

void TimeBasedAlarmInterface_Init(void);
void TimeBasedAlarmInterface_Shutdown(void);
```

Code Discussion

These files are in the operations training and can be copied and pasted into your project to provide access to the following variables for the DOF interface that you will need in requestor and provider code:

The DOFInterface instance. The variable to use when you need a DOFInterface instance is TBAInterface_DEF.

The DOFInterfaceID instance. The variable to use when you need a DOFInterfaceID instance is TBAInterface_IID. A DOFInterfaceID instance can also always be obtained from the DOFInterface instance using the *DOFInterface_GetInterfaceID(DOFInterface)* function. Samples in this guide do not show this.

Interface item identifiers. The header file includes #defines for all of the interface item identifiers shown in the table earlier. We will use these variables in sample code to avoid passing in magic numbers.

Chapter 2: Basic Setup

Before you can begin writing operations, you must set up the basic code that makes an application capable of being either a provider or a requestor. Both providers and requestors need an instance of DOFObject to initiate or process operations involved with interface functionality. Instantiating the DOFObject class requires that you first have an instance of two other classes: DOFSystem and DOFObjectID. (This is covered in later sections.)

If you want to use the operations training code to set up a working environment, do the following:

1. Copy the following files from any of the trainings into your project:
   - DOFTrainingDOFAbstraction.c
   - DOFTrainingDOFAbstraction.h
   - TimeBasedAlarmInterface.c
   - TimeBasedAlarmInterface.h
2. Include the header files in your requestor and provider applications.
3. Call *DOFTrainingAbstraction_Init(void)* and *TimeBasedAlarmInterface_Init(void)* in an initialization routine.
4. Call *DOFTrainingAbstraction_Shutdown(void)* and *TimeBasedAlarmInterface_Shutdown(void)* in a shutdown routine.

Initializing the Library

Before you can execute any DOF code in the C OAL, you must initialize the library.

To initiate the OAL, include the following call at the beginning of your *main* function:

```
DOF_Init();
```

When you are finished using the C OAL, you must shut down the library by calling the following at the end of your *main* function:

```
DOF_Shutdown();
```

Obtaining DOFSystem Instances

To create a DOFObject, you must use a method of DOFSystem, so you need an instance of DOFSystem on which to call the method. In general, creating DOFSystem instances is a concern for system developers. This guide doesn't cover the creation of DOFSystem instances, but assumes that a system developer will pass it to you or will have created a class with a method in it for creating the system.

If you want to use the operations training code described in this guide, there is a DOF abstraction header file that has a method in it for creating a system, which you can use in your project to get a DOFSystem instance.

If you copy DOFTrainingDOFAbstraction.c and DOFTrainingDOFAbstraction.h into your project and include the header file, you can use *DOFTrainingAbstraction_CreateSystem(void)* to get an instance of DOFSystem. You must destroy the DOFSystem instance in your shutdown routine. Call *DOFSystem_Destroy(DOFSystem)*.

Sample code in this guide assumes the DOFSystem instance has already been obtained; those instances are referred to in this guide by the variable name *system*.

Note: Both a DOF and a DOFSystem are required to create a DOFObject. This guide assumes you are using the DOF abstraction code in the training, which creates a DOF for you. For more information about DOF or DOFSystem, refer to the *DOF Connectivity Programmer's Guide* in the DOF Essentials SDK.

Creating DOFObjectID Instances

The DOFObjectID class in the DOF libraries is used to create unique object identifiers (OIDs) for any type of object that needs one. In the simple scenario of a single provider and a single requestor, the OID identifies the provider:

- The provider creates a DOFObjectID using an OID that identifies itself as a DOF object in the DOF Object Model. Its DOFObjectID instance essentially states, "I am this object."
- The requestor uses the *same* OID to create its DOFObjectID. Its DOFObjectID instance essentially states, "This is the provider I am going to want to request functionality from."

In more advanced scenarios, each requestor might need its own OID. For example, many services are essentially provider applications capable of creating multiple DOFObjects that provide the same functionality for multiple requestors, while maintaining separate state or database information for each individual requestor.

In a production environment, OIDs should be carefully managed and assigned to objects, and the individual in charge of managing the OIDs may tell you what to use; however, you may want to create your own. *Appendix A: Formatting OIDs* contains directions on how you can create and properly format an essentially unlimited number of unique OIDs if you own your own registered Internet domain or how you can use GUIDs to create OIDs.

Sample Code

The C OAL contains multiple functions for instantiating DOFObjectID. The following example shows how to use an OID class constant and a string in the *DOFObjectID_Create_String(DOFObjectIDClass, const char *)* function:

```
static DOFObjectID providerID = DOFObjectID_Create_String
         (DOFOBJECTIDCLASS_EMAIL, "provider@opendof.org");
```

You must destroy the DOFObjectID when you are finished with it:

```
DOFObjectID_Destroy(providerID);
```

Creating DOFObject Instances

The DOFObject class essentially represents a provider on the network. Even in requestor code, it is conceptualized as the provider you want to talk to. In the DOF Object Model, the DOFObject instance is the DOF object. The methods that enable you to initiate and process operations are contained in this class. In addition, the DOFObject is used by the internal library for addressing and routing. The create methods for DOFObject require an instance of DOFObjectID.

Sample Code

The following example shows how to instantiate DOFObject:

```
static DOFObject provider = DOFSystem_CreateObject(system, providerID);
```

You must destroy the DOFObject when you are finished with it. The following shows how to destroy DOFObject instances:

```
DOFObject_Destroy(provider);
```

Code Discussion

If you are writing a requestor application, you are ready to begin writing methods that initiate operations once you have a DOFObject instance. However, if you are writing a provider application, more preparation is required, as described in the next section.

Making a DOFObject a Provider

For a DOFObject to act as a provider, you must implement an interface that enables the DOFObject to listen for and respond to operations from requestors. This section describes only how to set up the interface implementation. Later sections of the guide describe how to implement the callbacks in the interface.

Implementing the Provider Interface

To fully implement the DOFObjectProvider interface, you declare all the functions used in the interface, and then place them in a struct.

Sample Code

The following sample shows a full implementation of DOFObjectProvider:

```
static void complete ( DOFObjectProvider self, const DOFOperation operation,
         const DOFException except );
static void invoke ( DOFObjectProvider self, const DOFOperation operation,
         DOFRequest request, const DOFInterfaceMethod method,
         uint16 parameterCount, const DOFValue parameters[] );
static void get ( DOFObjectProvider self, const DOFOperation operation,
         DOFRequest request, const DOFInterfaceProperty property );
static void set ( DOFObjectProvider self, const DOFOperation operation,
         DOFRequest request, const DOFInterfaceProperty property,
         const DOFValue value );
```

```
static void subscribe ( DOFObjectProvider self, const DOFOperation operation,
         DOFRequest request, const DOFInterfaceProperty property,
         uint32 minPeriod);
static void subscribeComplete( DOFObjectProvider self,
         const DOFOperation operation, DOFRequest request,
         const DOFInterfaceProperty property);
static void register ( DOFObjectProvider self, const DOFOperation operation,
         DOFRequest request, const DOFInterfaceEvent evt);
static void registerComplete ( DOFObjectProvider self,
         const DOFOperation operation, DOFRequest request,
         const DOFInterfaceEvent evt);
static void session ( DOFObjectProvider self, const DOFOperation operation,
         DOFRequest request, const DOFObject object,
         const DOFInterfaceID interfaceID, const DOFObjectID sessionID,
         const DOFInterfaceID sessionType );
static void sessionComplete ( DOFObjectProvider self,
         const DOFOperation operation, DOFRequest request,
         const DOFObject object, const DOFInterfaceID interfaceID,
         const DOFObjectID sessionID, const DOFInterfaceID sessionType );

static const struct DOFObjectProviderFns_t providerFns = { complete, invoke, get,
         set, subscribe, subscribeComplete, register, registerComplete, session,
         sessionComplete};
static DOFObjectProvider_t ProviderListener = { &providerFns };
```

Code Discussion

Function names and variable names may be changed, but the functions must take the argument types shown in the example.

The library defines default behavior for each of the functions, so you can pass NULL for any of the functions you do not need.

Sample Code

The following code shows how you might implement the provider interface for a DOF interface that has a single read-only property and needs to support *get* and *subscribe* operations:

```
static void get ( DOFObjectProvider self, const DOFOperation operation,
         DOFRequest request, const DOFInterfaceProperty
         property );

static const struct DOFObjectProviderFns_t providerFns = { NULL, NULL, get, NULL,
         NULL, NULL, NULL, NULL, NULL, NULL};
static DOFObjectProvider_t ProviderListener = { &providerFns };
```

Initiating a Provide Operation

Later on, we will cover how to begin providing an interface in response to a demand (see *Chapter 8: Activate-Level Interest*); however, for functionality discussed in earlier chapters, you should program your provider to begin providing when it starts up.

The method needed to begin providing is *DOFObject_BeginProvide (DOFObject, DOFInterface, uint32, const DOFObjectProvider, void *)*.

Sample Code

The following is a sample of the *DOFObject_BeginProvide* function:

```
static DOFOperation provideOperation = DOFObject_BeginProvide(provider,
         TBAInterface_DEF, DOF_TIMEOUT_NEVER, &provider, NULL);
```

The provide operation should also be canceled and must be destroyed, in this case, in your shutdown routine. Call the following to cancel and destroy the operation:

```
DOFOperation_Cancel (provideOperation);
DOFOperation_Destroy (provideOperation);
```

Code Discussion

The following describes the parameters of the *DOFObject_BeginProvide* function:

DOFObject. The first parameter is the DOFObject instance created earlier. Because the C OAL has an object-oriented model, but C is not an object-oriented language, the first parameter of many functions in the C OAL is an instance of the "class" the function belongs to.

DOFInterface. This parameter is the DOFInterface instance that was described in the section on *The Time-Based Alarm Interface*.

uint32. This parameter represents a "timeout." In the case of an asynchronous operation like provide, the timeout is how long the operation will remain active without completing and shutting down. We want our provider to continue providing for as long as it is on, so we used the "timeout never" constant provided by the library.

const DOFObjectProvider. This argument must be an instance of the DOFObjectProvider_t struct.

void *. This parameter offers additional flexibility in the library. You can think of it as an "attachment" that your provider application would receive every time the callbacks in the provider interface are called. So if you wanted to receive custom information in all of the provider interface callbacks, you would create data (most likely a struct), cast it as a void pointer, and pass it in this argument. This is an advanced and highly customized use of the library, so the rest of this guide ignores "context" parameters. In our samples, we pass null arguments.

The *DOFOperation_Cancel* call triggers a call to the provider's *complete* function (the first function in the provider interface); it is often a best practice to include the destroy call in that callback. This practice sets the stage for more advanced uses of the library where providers begin providing their interfaces on demand, and provide operations may be canceled through multiple functions. If you do not get into the habit of destroying your provide operation this way, it can be easy to create code that attempts to destroy the DOFOperation multiple times.

Default Provider Behavior

Later on, we show you how to implement functionality in each of the methods or functions in the provider interface; nevertheless, at this point it is useful to be aware of the default behavior that occurs if you pass NULL for the functions.

Get, Set, Invoke, and Session Calls

When the library receives *get*, *set*, and *invoke* operations, it first checks these operations against the DOF interface definition. If the interface does not contain a corresponding property or method, the provider application's method is never called. This is according to the following specifications:

- Get is never called if the DOF interface defines no readable properties.
- Set is never called if the DOF interface defines no writable properties.
- Invoke is never called if the DOF interface defines no methods.

If the interface *does* define properties or methods that correspond with these operations, but you do not override the *get*, *set*, or *invoke* methods, the OAL returns an "application error" exception. The provider, however, is in breach of its contract. A provider application should always override the default behavior of the *get*, *set*, or *invoke* callbacks if the DOF interface defines corresponding items of functionality.

Note: The default behavior for the session method is to return a "not supported" exception. Providers should implement session when they provide an interface that is defined as a base interface for sessions. (A discussion about the session method is beyond the scope of this guide.)

Subscribe and Register Calls

As with the get, set, and invoke operations, the subscribe and register functions are not called under the following conditions:

- Subscribe is never called if the DOF interface defines no readable properties.
- Register is never called if the DOF interface defines no events.

However, the default behavior in these methods is to automatically accept the subscribe or register operation. So, unlike with *get*, *set*, and *invoke*, the provider does not need to implement additional functionality in the *subscribe* and *register* callbacks to support subscribe and register operations.

Subscribe, Register, and Session Complete Calls

These callbacks are useful places to clean up any resources allocated in association with subscribe, register, or session operations. The default behavior in these callbacks is to do nothing, but you should override the default behavior if you need to perform cleanup associated with the operations.

Complete Call

This method is called when a provide operation completes, and it can be used to clean up resources. You can also use it to destroy the provide operation itself. The default behavior for this callback is to do nothing, but you should override the default behavior if you need to clean up resources associated with the provide operation.

Note: If the *subscribeComplete*, *registerComplete*, or *sessionComplete* callbacks contain functionality associated with cleaning up resources, it is a good practice to also clean up those resources in the *complete* callback for the provide operation. The library cannot call *subscribeComplete*, *registerComplete*, or *sessionComplete* after *complete* has been called for the provide operation, and it is possible for these calls to be delivered out of order.

Chapter 3: Synchronous Operations

The OAL provides synchronous operations for getting or setting a DOF interface property value and for invoking a DOF interface method. In this chapter, you will learn how to implement these three types of operations for both providers and requestors.

Getting Properties Synchronously

A requestor initiates a synchronous get operation using the *DOFObject_Get* function of the DOFObject class. To properly return a value to the requestor, the provider must override the *get* callback of the DOFObjectProvider interface. The sequence of a synchronous get operation is shown in the following diagram.

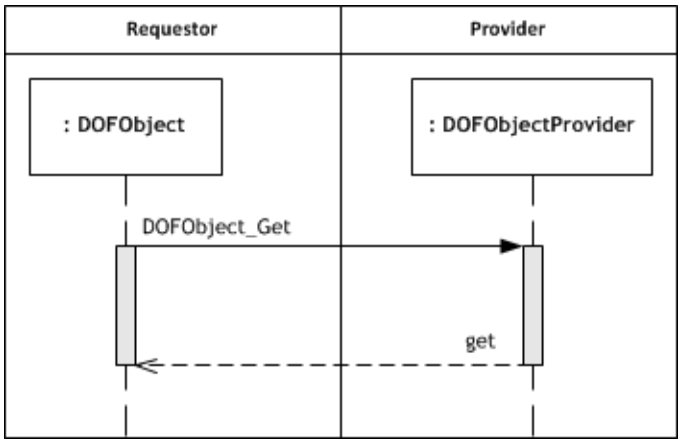

Figure 1. Synchronous Get Operation Sequence Diagram

Provider

For a provider to handle any interface property, it must store the property's value, usually in a global variable. The code samples shown in this guide use the two properties of the Time-Based Alarm interface described in the introduction. The type definition for the AlarmActive property is a boolean, so you can assume when looking at the samples that earlier code has initialized a boolean called *isActive* and set its initial value to false. For the AlarmTime value, assume a DateTime was initialized with the variable name *alarmTime* and set to zero. It is often good practice to set the global variables for properties to static and limit them to file scope.

With the property initialized, you can now create the functionality that will enable the provider to return the value of that property to requestors.

Sample Code

The following sample shows how to implement the *DOFObjectProvider::DOFObjectProviderFns_t::Get* function to return the properties in the Time-Based Alarm interface:

```
static void get(DOFObjectProvider self, const DOFOperation operation,
         DOFRequest request, const DOFInterfaceProperty property){
   DOFItemID requestItemID = DOFInterfaceProperty_GetItemID(property);
   if(requestItemID == TimeBasedAlarm_AlarmActiveID){
      DOFValue dofBoolean = DOFValueBoolean_Create(isActive);
      DOFRequestGet_Return(request, dofBoolean);
      DOFValue_Destroy(dofBoolean);
   } else if(requestItemID == TimeBasedAlarm_AlarmTimeValueID) {
      DOFValue dofDateTime = DOFValueDateTime_Create(alarmTime);
      DOFRequestGet_Return(request, dofDateTime);
      DOFValue_Destroy(dofDateTime);
   }
}
```

Code Discussion

The following steps provide directions for coding a provider's *get* callback, using the sample above for an example:

1. Compare the value of the item identifier passed to you with the item identifiers defined in the DOF interface.
   Obviously, an interface definition can contain more than one of any item type, so you should compare what was passed to you to each of the interface properties to determine which value to return. In addition, it is a good practice to make this check even when you have only one item of a type defined in an interface.
2. Create a DOFValue instance from the stored property value.
   - A DOFValue instance is what the library needs to return to the requestor, so the first step is to create a DOFValue instance from a stored primitive type. The C OAL has separate create functions for each primitive type, which all return a DOFValue. For a full listing of all DOFValue create functions, see the API reference documentation.
   - Some types defined in DOF interfaces, such as strings and blobs, may require additional parameters to construct, such as encoding and maximum length for strings. Each DOF interface defines the valid values or ranges for these types.
3. Use a DOFRequest method to return the DOFValue instance to the requestor.
   - The library passes a DOFRequest object to the provider in the *get* callback. Call the function that properly corresponds with the operation type—in this case, *DOFRequestGet_Return(DOFRequest, DOFValue)* and pass the DOFRequest that was passed to you and the DOFValue created in step 1 as arguments in this function.
   - Whenever you implement the *get* callback, you *must* call either *DOFRequestGet_Return* or *DOFRequestGet_Throw*. If you do not, the requestor's operation will time out. In addition, the DOFRequest will not be destroyed, causing memory leaks. Usage of *DOFRequestGet_Throw* is shown in *Chapter 4: Exception Handling*.
4. Destroy the DOFValue.
   Destroy it by calling *DOFValue_Destroy(DOFValue)*.

Requestor

To get the value of a property, requestors need to use the *DOFObject_Get* function. The function prototype is *DOFObject_Get (DOFObject, const DOFInterfaceProperty, DOFOperationControl, uint32, DOFException *)*.

Note: Using an operation control is beyond the scope of this guide. Samples shown in this guide always pass NULL for the DOFOperationControl argument.

In the next chapter, we discuss how to add exception handling, but in this chapter, the samples pass NULL for the DOFException pointer.

Sample Code

The following sample shows how to use the *DOFObject_Get* function to get the value of the AlarmActive property:

```
static boolean Requestor_sendGetRequest(void){
   DOFInterfaceProperty property = DOFInterface_GetProperty (TBAInterface_DEF,
            TimeBasedAlarm_AlarmActiveID);

   DOFResult result;
   DOFValue value;
   boolean unwrappedResult;

   result = DOFObject_Get(provider, property, NULL, 10000, NULL);
   value = DOFResult_GetValue(result);
   unwrappedResult = DOFValueBoolean_Get(value);

   DOFResult_Destroy(result);
   return unwrappedResult;
}
```

Code Discussion

The following steps provide directions for coding your own requestor's *get* method, using the samples above for an example:

1. Get the property from the interface.
   The library provides a function for getting the property from a DOFInterface instance. This is *DOFInterface_GetProperty (DOFInterface, DOFItemID)*, where the DOFItemID parameter is a uint16 that represents the item identifier. The sample passes the variable defined in TimeBasedAlarmInterface.h.
2. Call the requestor's DOFObject_Get function.
   - This returns a DOFResult, so use it to initialize an instance of DOFResult. Pass the property created in Step 1 to this method.
   - The uint32 parameter represents a timeout, which is a time in milliseconds that the application will wait to receive the result of the operation from the property. Because this is a blocking call, you probably do not want to set the timeout too high, but it should be long enough for the requestor to communicate with the provider across the network. The samples have set the timeout to 10 seconds, but you may need to adjust this to a longer time if your network communication is slow. You would also want to use a longer

time in cases where the property value is a large chunk of data, such as a large blob.

Note: The DOFResult class has a method that returns the DOFValue. This is *DOFResult_GetValue(DOFResult)*.

3. Unwrap the DOFValue.
   Use a get function that corresponds to the type of primitive you need to retrieve. For example, our sample uses *DOFValueBoolean_Get* to retrieve a boolean, because the AlarmActive property is defined as a boolean in the DOF interface. If, for example, the DOF interface property were a uint8, you would need to use *DOFValueUInt8_Get*. (Refer to the API reference documentation for more information.)
4. Destroy the DOFResult.
   You must destroy the DOFResult by calling *DOFResult_Destroy*. Unlike in the sample code for a provider, you do not need to destroy DOFValue. The requestor received the DOFValue in the DOFResult struct returned by *DOFObject_Get*, and the DOFValue is destroyed when the DOFResult struct is destroyed.

Setting Properties Synchronously

A requestor initiates a synchronous *set* operation using the *DOFObject_Set* function of the DOFObject class. To properly return a value to the requestor, the provider must implement the *set* method in the DOFObjectProvider interface. The sequence of a synchronous set operation is shown in the following diagram.

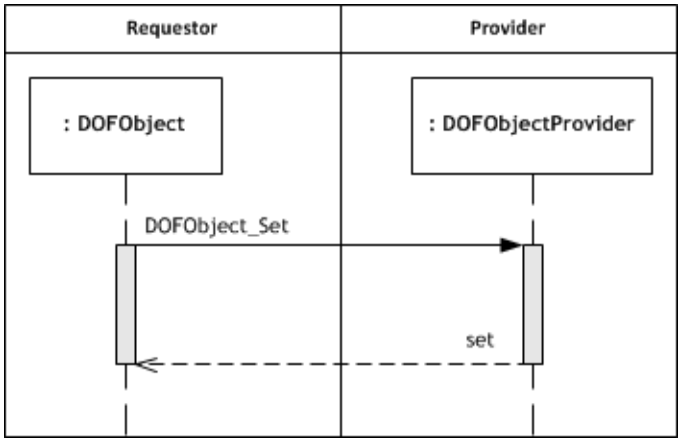

Figure 2. Synchronous Set Operation Sequence Diagram

Provider

The provider often needs to implement functionality in response to a *set* operation, such as changing the physical state of a device or triggering user notifications. Such application logic is specific to the device. For example, our sample code shows how to set the AlarmActive property in the Time-Based Alarm interface. This property determines whether or not the AlarmTriggered event will be triggered at the time stored in the AlarmTimeValue property. Application logic requires changing the stored value of the property in response to a set operation.

Outside of application logic, DOF requires only that the provider respond to the requestor.

Sample Code

The following sample shows how to implement the *DOFObjectProvider::DOFObjectProviderFns_t::Set* function to return the value of the AlarmActive property:

```
static void set( DOFObjectProvider self, const DOFOperation operation,
         DOFRequest request, const DOFInterfaceProperty property,
         const DOFValue value ){
   DOFItemID requestItemID = DOFInterfaceProperty_GetItemID(property);
   if(requestItemID == TimeBasedAlarm_AlarmActiveID){
      isActive = DOFValueBoolean_Get(value);
      DOFRequestSet_Return(request);
   }
}
```

Code Discussion

The following steps provide directions for coding a provider's *set* method, using the samples above for an example:

1. Unwrap the DOFValue and set the stored property value equal to the unwrapped value.
2. Use a DOFRequest method to respond to the requestor.
   Although no values need to be returned, you must call *DOFRequestSet_Return* or *DOFRequestSet_Throw* to send an acknowledgment to the requestor and prevent its operation from timing out.

Requestor

To set the value of a property, requestors need to call *DOFObject_Set (DOFObject, const DOFInterfaceProperty, DOFValue, DOFOperationControl, uint32, DOFException *).*

Sample Code

The following sample shows how to use the *DOFObject_Set* function to set the value of the AlarmActive property:

```
static void sendSetRequest(boolean isActive){
   DOFInterfaceProperty property = DOFInterface_GetProperty (TBAInterface_DEF,
            TimeBasedAlarm_AlarmActiveID);

   DOFValue setValue;
   DOFResult setResult;

   setValue = DOFValueBoolean_Create(isActive);
   setResult = DOFObject_Set(provider, property, setValue, NULL, 10000, NULL);

   DOFValue_Destroy(setValue);
   DOFResult_Destroy(setResult);
}
```

Code Discussion

The following steps provide directions for coding a requestor's *set* method, using the samples above for an example:

1. Get the property from the interface.
2. Convert the primitive input type to a DOFValue.
3. Call the requestor's *DOFObject_Set* function and pass it the new value to be set.
   - As with the get operation, the uint32 parameter is a timeout that represents how long you want the application to wait to complete.
   - Now that we have seen code for both get and set operations, we are ready to discuss the DOFResult class. All of a requestor's methods for initiating synchronous operations return a DOFResult; however, each DOFResult contains slightly different information.
   - The returned DOFResult is a struct that contains either a DOFValue (get operations), a DOFValueList (invoke operations), or NULL (set operations). You must initialize the DOFResult, even for set operations, because it needs to be destroyed. However, calling *DOFResult_GetValue* on the DOFResult returned from a set operation returns NULL.
4. Destroy the DOFValue and the DOFResult.
   In the requestor's get operation, the DOFValue was part of the DOFResult struct and did not need to be destroyed separately; however, in this code for the set operation, we created the DOFValue and both the DOFValue and DOFResult need to be destroyed.

Invoking Methods Synchronously

A requestor initiates a synchronous invoke operation using the *DOFObject_Invoke* function of the DOFObject class. To properly implement an invoke operation, the provider must use the *invoke* method in the DOFObjectProvider interface to execute the functionality defined for the method in the DOF interface and return any required values to the requestor. The sequence of a synchronous invoke operation is shown in the following diagram.

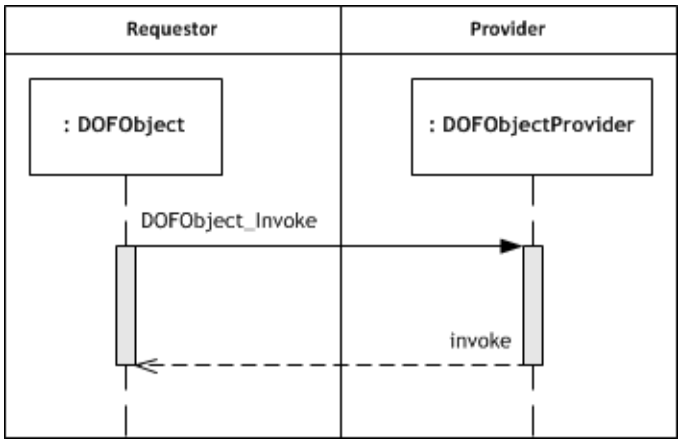

Figure 3. Synchronous Invoke Operation Sequence Diagram

Provider

Because DOF interface methods are methods in a traditional programming sense, implementing the functionality they define can be as complex or simple as implementing any programming method. The SetNewTime method in our example Time-Based Alarm interface is a relatively simple one: set the AlarmTime property and return the value of the AlarmActive property.

You may wonder why an interface designer would make a method like this one rather than just making the AlarmTime property writable and allowing the requestor to set it. However, this is actually a fairly common type of method in DOF interfaces because it allows the separation of permissions in the DOF security model. Many requestors could be granted write permission for the interface, and they could then set the AlarmActive property, but they would still be unable to change the read-only AlarmActive property unless they also had execute permission, which would allow them to invoke the method.

For a provider to handle any interface property, it must store the property's value, usually in a global variable. The code samples shown in this use the AlarmActive property of the Time-Based Alarm interface described in the introduction. The type definition for the AlarmActive property is a boolean, so you should assume when looking at the samples that earlier code has initialized a boolean called *isActive* and set its initial value to false. It is often good practice to set the global variables for properties to static and limit them to file scope.

The provider needs to store the value of the second interface property in a global variable. You should assume when looking at the samples that earlier code has initialized a DateTime called *alarmTime* and set its initial value to null.

Sample Code

The following sample shows how to implement the *DOFObjectProvider::DOFObjectProviderFns_t::Invoke* function to perform the functionality defined for the SetNewTime method:

```
static void invoke( DOFObjectProvider self, const DOFOperation operation,
         DOFRequest request, const DOFInterfaceMethod method,
         uint16 parameterCount, const DOFValue parameters[] ){
   DOFItemID requestItemID = DOFInterfaceMethod_GetItemID(method);
   if(requestItemID == TimeBasedAlarm_SetNewTimeID){
      DOFValue inputParameter;
      DateTime unwrappedInputParam;
      DOFValue returnValue;
      DOFValue returnValues[];

      /* Handling Method Input */
      inputParameter = parameters[0];
      unwrappedInputParam = DOFValueDateTime_Get(inputParameter);
      alarmTime = unwrappedInputParam;

      /* Sending Method Output */
      returnValue = DOFValueBoolean_Create(isActive);
      returnValues[0] = returnValue;
      DOFRequestInvoke_Return(request, 1, returnValues);
```

```
        DOFValue_Destroy(returnValue);
    }
}
```

Code Discussion

The following steps provide directions for coding a provider's *invoke* method, using the samples above for an example:

1. Get the DOFValue instances that represent the input parameters out of the array.
   - When methods have no input parameters, you can skip steps 1 through 3.
   - Because a method can have multiple input parameters, input parameters are always passed to the invoke method as an array, even if the method has zero or one input parameters.
   - Because our sample method has only one input parameter, we know it is at offset zero and so we retrieve it and use it to initialize a DOFValue instance. When DOF interface methods have multiple input parameters, they must be ordered as the interface defines.
2. Unwrap the DOFValue.
3. Use input parameters as defined by the DOF interface.
   For our sample interface, we are setting the alarmTime variable equal to the unwrapped input parameter, although functionality may be more complex for many interface methods.
4. Create DOFValue instances for the output parameters.
   - All output parameters must be DOFValue instances. Because our sample method has only one output parameter, the sample code creates a single DOFBoolean.
   - When methods have no output parameters, you can skip steps 5 and 6.
5. Add the DOFValue instances to an array.
   Create a DOFValue array and populate it with the DOFValue instances that represent your output parameters. If you have multiple output parameters, you must provide them in the order defined by the DOF interface.
6. Use a DOFRequest method to respond to the requestor.
   - Use *DOFRequestInvoke_Return (DOFRequest, uint16, const DOFValue[])*. The uint16 parameter is the number of items in the DOFValue array.
   - Even when the method has no output values, you *must* call *DOFRequestInvoke_Return* or *DOFRequestInvoke_Throw* to prevent the requestor from timing out.
7. Destroy all created DOFValue instances.
   DOFValue instances that were passed to you in the DOFRequest do not need to be destroyed. DOFValue instances you created do need to be destroyed.

Requestor

To invoke a method, requestors need to call *DOFObject_Invoke*. The function prototype is *DOFObject_Invoke (DOFObject, const DOFInterfaceMethod, uint16, const DOFValue [], DOFOperationControl, uint32, DOFException *)*.

The uint16 parameter is a count of items in the DOFValue array. As in similar methods, the uint32 parameter represents the timeout.

Sample Code

The following sample shows how to use the *DOFObject_Invoke* function to invoke the SetNewTime method:

```
boolean Requestor_sendInvokeRequest(DateTime alarmTime){
   DOFInterfaceMethod method = DOFInterface_GetMethod(TBAInterface_DEF,
            TimeBasedAlarm_SetNewTimeID);
   DOFValue inputParameter;
   DOFValue parameterArray[1];
   DOFResult result;
   const DOFValue * returnValues;
   boolean unwrappedReturnVal;

   /* Sending Method Input */
   inputParameter = DOFValueDateTime_Create(alarmTime);
   parameterArray[0] = inputParameter;
   result = DOFObject_Invoke(provider, method, 1, parameterArray, NULL, 10000,
            NULL);

   DOFValue_Destroy(inputParameter);

   /* Handling Method Output */
   returnValues = DOFResult_GetValueList(result);
   unwrappedReturnVal = DOFValueBoolean_Get(returnValues[0]);

   DOFResult_Destroy(result);

   return unwrappedReturnVal;
}
```

Code Discussion

The following steps provide directions for coding a requestor's *DOFObject_Invoke* function, using the samples above for an example:

1. Get the method from the interface.
2. Convert the input parameter (if present) to a DOFValue instance.
   If the methods have no input parameters, skip step 3.
3. Add the DOFValue instance to an array.
   You must create a DOFValue array and populate it with the DOFValue instances that represent your input parameters. If you have multiple input parameters, you must provide them in the order defined by the DOF interface.
4. Call the requestor's *DOFObject_Invoke* function.
   When methods have no output parameters, you can skip steps 5 through 7.
5. Get the array of output parameters from the DOFResult.
6. Get the DOFValue instance that represents the output parameter from the array.
7. Unwrap the DOFValue.
8. Destroy the DOFResult and the created DOFValue.
   DOFValue instances created for input parameters should be destroyed. Output parameters are destroyed along with the DOFResult struct.

Chapter 4: Exception Handling

For simplicity, the operations shown in the previous section did not do exception handling. In a production environment, you should never perform these operations without proper exception handling.

This section covers the following topics:

- DOF exception types
- Modifying provider code to *throw* library exceptions
- Modifying requestor code to *catch* library exceptions

DOF Exception Types

The DOF OALs define two basic exception types:

DOFProviderException. This class represents exceptions defined by a DOF interface.

DOFErrorException. This class represents enumerated exceptions defined in the DOF library.

Returning Exceptions in Provider Applications

This section provides directions for creating provider and error exceptions and returning them to a requestor.

Provider Exceptions

The sample code in this section shows the provider's *invoke* method that was shown in the previous chapter. It has been modified to check for the condition that should cause a provider exception to be thrown and to throw the exception when that condition occurs. The discussions after the samples cover only the new information.

The signature for the function to create provider exceptions is *DOFProviderException_Create(DOFInterfaceException, uint16, const DOFValue *)*.

Sample Code

The following sample shows how to add a provider exception to the provider's *invoke* method:

```
static void invoke( DOFObjectProvider self, const DOFOperation operation,
         DOFRequest request, const DOFInterfaceMethod method,
         uint16 parameterCount, const DOFValue parameters[] ){
   DOFItemID requestItemID = DOFInterfaceMethod_GetItemID(method);
   if(requestItemID == TimeBasedAlarm_SetNewTimeID){
      DOFValue inputParameter;
      DateTime unwrappedInputParam;

      /* Handling Method Input */
```

```
        inputParameter = parameters[0];
        unwrappedInputParam = DOFValueDateTime_Get(inputParameter);

        if(unwrappedInputParam < PCRTime_GetMS()){
           DOFInterfaceException interfaceException;
           DOFException ex;

           interfaceException = DOFInterface_GetException
                    (TBAInterface_DEF, TimeBasedAlarm_BadTimeValueID);
           ex = DOFProviderException_Create(interfaceException, 0, NULL);
           DOFRequest_Throw(request, ex);

           DOFException_Destroy(ex);
        }else{
           DOFValue returnValue;
           DOFValue returnValues[];

           alarmTime = unwrappedInputParam;

           /* Sending Method Output */
           returnValue = DOFValueBoolean_Create(isActive);
           returnValues[0] = returnValue;
           DOFRequestInvoke_Return(request, 1, returnValues);

           DOFValue_Destroy(returnValue);
        }
     }
}
```

Code Discussion

The following steps provide directions for creating a provider exception, using the samples above for an example:

1. Check for the error condition.
   In the Time-Based Alarm interface, the exception should be thrown if the time provided by the requestor is earlier than the current time. The code uses an "if" statement to check for this condition.
2. Get the exception from the interface.
3. Create DOFValue instances for the output parameters, if any.
   When exceptions have no output parameters, you can skip step 4. The samples do not show these steps because the BadTimeValue exception defined in the Time-Based Alarm interface has no output parameters.
4. Add the DOFValue instances to an array.
   Output parameters should be provided in the order that is defined by the DOF interface.
5. Create a DOFProviderException.
   Use the function shown in the introduction to this section. The uint16 parameter is a count of the items in the array. If the exception has no output parameters, it must be zero.
6. Use a DOFRequest method to respond to the requestor.

The *DOFProviderException_Create* function returns a DOFException, which you then pass to *DOFRequest_Throw(DOFException)*.

7. Destroy the DOFException.

Error Exceptions

The code samples in this section show the same *get* operations that were shown earlier in this chapter, but here they have been modified to throw an exception as a default case. Because this code should never be reached, you could use an assert in this situation, but the exception is also a logical default case. The directions after the samples only cover the new information.

The function to create a DOFErrorException is *DOFErrorException_Create(DOFError, const char *)*.

Sample Code

The following sample shows how to add an error exception to the provider's *get* method:

```
static void get(DOFObjectProvider self, const DOFOperation operation,
         DOFRequest request, const DOFInterfaceProperty property){
   DOFItemID requestItemID = DOFInterfaceProperty_GetItemID(property);
   if(requestItemID == TimeBasedAlarm_AlarmActiveID){
      DOFValue dofBoolean = DOFValueBoolean_Create(isActive);
      DOFRequestGet_Return(request, dofBoolean);
      DOFValue_Destroy(dofBoolean);
   } else if(requestItemID == TimeBasedAlarm_AlarmTimeValueID) {
      DOFValue dofDateTime = DOFValueDateTime_Create(alarmTime);
      DOFRequestGet_Return(request, dofDateTime);
      DOFValue_Destroy(dofDateTime);
   } else {
      DOFException ex;
      ex = DOFErrorException_Create(DOFERROR_NOT_SUPPORTED, "This provider does
               not support the specified interface item.\n");
      DOFRequest_Throw(request, ex);
      DOFException_Destroy(ex);
   }
}
```

Code Discussion

The following steps provide directions for creating an error exception, using the samples above for an example:

1. Create a DOFErrorException.
   Use the *create* function shown in the introduction to this section. The DOFError parameter represents an error code. The DOFError module contains #defines for the available error codes.
2. Use a DOFRequest method to respond to the requestor.
   The *DOFErrorException_Create* function returns a DOFException, which you then pass to *DOFRequest_Throw(DOFException)*.
3. Destroy the DOFException.

Catching Exceptions in Requestor Applications

All of the methods in the DOF OALs that initiate operations are capable of throwing exceptions, so requestors should be prepared to catch exceptions whenever they initiate operations. In addition, requestors need to be aware of the provider exceptions defined in DOF interfaces and the conditions that trigger them so they can handle them correctly.

The exceptions defined by the OALs do not cover all of the error conditions that can occur in code. In general, the exceptions defined in the OALs are those that can occur in communication between requestors and providers. Programmers should handle general programming errors according to best practices.

C programmers who are unfamiliar with how object-oriented programming languages handle errors may benefit from doing additional research on the topic. However, the following list summarizes the way exception handling works in the C OAL:

- Errors are defined in a DOFException struct.
- OAL functions that may throw exceptions take a DOFException pointer argument.
- If you provide this argument, the function initializes the DOFException struct if an error condition occurs.
- A DOFException can be either a
  - DOFErrorException (DOFEXCEPTIONTYPE_ERROR)
  - DOFProviderException (DOFEXCEPTIONTYPE_PROVIDER)
- A DOFErrorException type is associated with a DOFError type definition that defines all of the error codes. (The codes are the same as those defined in three of the Java OAL classes: DOFException, DOFErrorException, and DOFSecurityException.)
- The DOFException struct contains functions for getting the error codes and messages associated with a DOFErrorException.
- For a DOFProviderException, the DOFException struct contains functions for getting output parameters that may be defined in a DOF interface.

Sample Code

The following sample shows how to add exception handling to the requestor's *invoke* method that was shown in the previous chapter:

```
static boolean Requestor_sendInvokeRequest(DateTime alarmTime){
   DOFInterfaceMethod method;
   DOFValue inputParameter;
   DOFValue parameterArray[1];
   DOFResult result;
   const DOFValue * returnValues;
   boolean unwrappedReturnVal;
   DOFException ex;

   /* Sending Method Input */
   method = DOFInterface_GetMethod(TBAInterface_DEF,
            TimeBasedAlarm_SetNewTimeID);
   inputParameter = DOFValueDateTime_Create( alarmTime );
   parameterArray[0] = inputParameter;
```

```
    result = DOFObject_Invoke(provider, method, 1, parameterArray, NULL, 10000,
             NULL);

    DOFValue_Destroy(inputParameter);

    if(!ex){
       /* Handling Method Output */
       returnValues = DOFResult_GetValueList(result);
       unwrappedReturnVal = DOFValueBoolean_Get(returnValues[0]);

       DOFResult_Destroy(result);

       return unwrappedReturnVal;
    } else {
       if(DOFException_GetType(ex) == DOFEXCEPTIONTYPE_ERROR){
          DOFError errorCode = DOFErrorException_GetError(ex);
          printf("Exception: %s %s\n", DOFError_GetString(errorCode),
                    DOFErrorException_GetMessage(ex));
       } else {
          DOFInterfaceException exception;
          DOFItemID itemID;

          exception = DOFProviderException_GetInterfaceException(ex);
          itemID = DOFInterfaceException_GetItemID(exception);

          if(itemID == TimeBasedAlarm_BadTimeValueID) {
             printf("Received BadTimeValue exception.\n");
          } else {
             printf("Received unknown provider exception. Item ID: %d\n", itemID);
          }
       }
       DOFException_Destroy(ex);
    }
    return FALSE;
}
```

Code Discussion

The following steps outline the changes that are shown in the code sample:

1. After using a method that initiates an operation, check whether the DOFException was filled.
2. If the exception was not filled, execute the code required to get the return value.
3. If the exception was filled, check which type of exception was returned (i.e., error or provider).
4. Handle the exception based on type.
5. Destroy the exception.

Chapter 5: Interest and Query Operations

Before we can talk about asynchronous operations, you need to understand how to set up interest and query operations. You will need interest and query operations if you want to program your requestor to work with multiple providers; performing an interest operation is also required before you can perform the operations described in *Chapter 7: Ongoing Asynchronous Operations*.

Interest Operations

Requestors perform interest operations for a number of reasons. The following use cases are discussed in this guide:

- To discover whether a specific provider is currently running and connected to the network
- To discover multiple providers of an interface
- To support operations that require interest
- To cause a provider that is capable of providing an interface on demand to begin providing it
- To cause datagram connections between a requestor and provider to be upgraded to streaming connections (**Note:** Further discussion of this scenario is beyond the scope of this guide.)
- To optimize network traffic by mapping a route between a requestor and provider, although this must be carefully balanced against the fact that interest operations flood the network and are stored in state on every network node. Use of interest operations where not otherwise required by an operation should be carefully managed by a system designer. (**Note:** Further discussion of this scenario is beyond the scope of this guide.)

Relationship Between Interest and Provide Operations

Interest is a requestor operation. The provider operation that corresponds with it is a provide operation. The following are true of interest and provide operations:

- A requestor sends an interest operation. Interest operations always flood the network and are stored in state on every node until the interest operation times out or is canceled.
- When a new node connects to a DOF network, the node it connects to automatically forwards its interest table so that new nodes will have all interest operations stored in state.
- Interest operations are sent every hour (by default) until they are canceled or time out.
- A provider stores its provide operation in its local state, but sends it across the network only when it receives a matching interest operation. Provide operations are routed directly to requestors that have sent corresponding interest operations.

Interest Levels

A requestor can express the following enumerated interest levels:

WATCH. This level of interest enables requestors to discover providers that are already providing an interface. It also causes direct routes to be set up between providers and interested requestors. (Covered in this chapter)

ACTIVATE. This level of interest includes all the functionality of the WATCH level. In addition, it causes providers that are capable of providing an interface but not already providing it to begin providing it.

CONNECT. This level of interest includes all the functionality of the ACTIVATE level. In addition, it causes streaming connections to be established between all nodes between the requestor and the provider, if those nodes are capable of establishing streaming connections. (Further discussion on the Connect interest level is beyond the scope of this guide.)

Query Operations

A requestor uses a query operation to parse the provides it receives in response to interest. It is important to remember that a query operation is a *local* operation. It is not sent to other nodes. It will not search the network for providers. It finds only provide operations that the requestor is storing in state on its local system. Because the system will not store provide operations without a matching interest operation, queries are useful only in conjunction with interest.

Discovering a Single Provider

One purpose of using watch-level interest to discover a specific provider is to ensure that the provider is available before you begin sending other operations across the network. The simplest way to do this is to send an interest operation and then call *DOFSystem_WaitProvider*. This method is a query operation shortcut. It returns a DOFObject representing a provider that matches the interest operation.

Expressing Interest

The full signature for the method used to express interest is *DOFSystem_BeginInterest (const DOFSystem, const DOFObjectID, const DOFInterfaceID, DOFInterestLevel, DOFOperationControl, uint32, const DOFSystemInterestCallback, void *).*

Sample Code

```
static DOFOperation interestOp;

...

interestOp = DOFSystem_BeginInterest(system, providerID, TBAInterface_IID,
         DOFINTERESTLEVEL_WATCH, NULL, DOF_TIMEOUT_NEVER, NULL, NULL);

...
```

```
DOFOperation_Cancel(interestOp);
DOFOperation_Destroy(interestOp);
```

Code Discussion

The following steps explain the sample code:

1. Declare a DOFOperation.
   Since the declared operation needs to be used in multiple blocks, it needs to be file scope or you need to pass pointers to it, where appropriate.
2. Call the *DOFSystem_BeginInterest* function.
   Although the sample does not show this, the begin interest call would likely be contained within a block, such as an initialization routine or other method. The following describes the parameters used in the sample:

   DOFSystem. We passed the DOFSystem instance obtained according to the instructions described in *Chapter 2: Basic Setup*.

   DOFObjectID. Pass a DOFObjectID created according to the instructions in *Chapter 2: Basic Setup*. You can use the library's DOFOBJECTID_BROADCAST constant to express interest in any provider of the specified interface. Sample code using this constant is shown later in this chapter under *Discovering Multiple Providers*.

   DOFInterfaceID. Pass a DOFInterfaceID instance. The one we passed was described in the chapter on the Time-Based Alarm interface. You can use the library's DOFINTERFACEID_WILDCARD constant to express interest in all interfaces provided by the provider with the specified DOFObjectID.
   Note: Using both the broadcast DOFObjectID and wildcard DOFInterfaceID creates a broad interest in any provider of any interface.

   DOFInterestLevel. Interest levels are enumerated. To initiate watch-level interest, the argument should be passed exactly as shown.

   DOFOperationControl. For the scope of this guide, pass NULL.

   uint32. For an interest operation, the timeout parameter does not represent how long you are willing to wait for results. Instead, it represents how long you want the interest operation to persist. For our purposes, we want the interest operation to last until we cancel it, so we used the "timeout never" constant defined by the library.

   DOFSystemInterestCallback. Interest listeners are handled at the DOF level and not covered in this guide. Pass a null argument.

   void *. This parameter takes a context argument. This is an advanced use case and is not covered in this guide. Pass a null argument.

3. Cancel the interest operation.

Although the sample does not show this, the interest operation would likely be canceled in a different block from where it is initiated.

4. Destroy the interest operation.
   You must destroy the interest operation, either directly after canceling it or in the complete call for an interest listener. if you had one.

The WaitProvider Query

The method described in this section is a shortcut for creating a full query. It blocks the requestor application and waits until an interest operation has received a provide that matches the query. It then returns a DOFObject for the provider. You would use this call to initialize an object instead of using the method call described in *Creating DOFObject Instances*.

The signature is *DOFSystem_WaitProvider (const DOFSystem, const DOFObjectID, uint32, const DOFInterfaceID[], uint32)*.

If you provide an array of multiple interface identifiers, the *DOFSystem_WaitProvider* function will not return a result unless the provider provides *all* of the interfaces in the query. You must have initiated multiple interest operations beforehand, corresponding to all the interfaces, or an interest operation that uses the wildcard DOFInterfaceID.

Sample Code

The following sample finds a provider of the Time-Based Alarm interface, corresponding with our earlier interest operation:

```
DOFInterfaceID iids[1];
iids[0] = TBAInterface_IID;

DOFObject provider = DOFSystem_WaitProvider(system, providerID, 1, iids, 10000);
```

Code Discussion

Build a DOFInterfaceID array and pass it to the function. Also pass an argument that represents the number of entries in the array. The final argument is a timeout. Because this is a blocking call, this timeout represents how long you want to wait for the query to complete. Because the query searches only the local results of interest operations, you do not need to consider network performance in this timeout.

DOFObject instances returned by the *DOFSystem_WaitProvider* function must still be destroyed, according to the instructions in *Creating DOFObject Instances*.

Customized Query Operations

To create a customized query operation, you need to do the following:

- Build a DOFQuery, which specifies what to query for
- Implement the DOFSystemQueryCallback interface, which enables you to handle the results of the query

In this section, we'll create a customized query that will allow us to discover multiple providers.

Building a Query

You use a builder pattern to create a DOFQuery instance. The DOF OALs commonly use a builder pattern for creating connectivity and other objects, but DOFQuery is the only object covered in this guide that uses the pattern. For DOFQuery, the following steps are involved in following the pattern:

- Instantiate DOFQueryBuilder.
- Use methods of the Builder to set parameters for the DOFQuery.
  Each of these methods returns a modified version of the Builder instance.
- Call the *build* method of the Builder, which returns a DOFQuery instance that matches the last configuration state of the Builder.

The following parameters can be set for a DOFQuery:

- One or more filters
- One or more restrictions
- A match style

The match style determines how OIDs with attributes will be handled in the query. This guide does not cover using OIDs with attributes, and this parameter will be left at its default value in the samples.

Filters

To create a useful query, you must add at least one filter to the builder. Filters determine which provide operations the query will search for. You can create filters that do the following:

- Search for a specific provider (based on OID) of a specific interface (based on IID)
- Search for a specific provider (based on OID) providing multiple specific interfaces (based on multiple IIDs)
- Search for a specific provider (based on OID) providing any interface
- Search for multiple specific providers (based on multiple OIDs) of one or more specific interfaces (based on IIDs)
- Search for any providers of a specific interface (based on IID)
- Search for any providers of multiple specific interfaces (based on multiple IIDs)

When results of a query are found, the *interfaceAdded* callback in the query operation listener is called (described later). In this scenerio, the following rules apply:

- If the filter contains only a single IID, the *interfaceAdded* callback will be called when a provider of that interface is found; however, if that provider also provides interfaces not included in the filter, the *interfaceAdded* callback will be called again for every interface that provider provides.
- If the filter contains multiple IIDs, the *interfaceAdded* callback will not be called unless a provider is found that provides *all* of the interfaces in the query. If such a

provider is found, the *interfaceAdded* callback will be called multiple times (once for each of the interfaces in the filter). If that provider provides interfaces other than those included in the filter, the *interfaceAdded* callback will also be called for all other interfaces that provider provides.

- If you specify both multiple providers and multiple interfaces in a filter, the results will be for *any* of the specified providers that provide *all* of the specified interfaces.

You can add as many filters as you want. The *interfaceAdded* callback is called whenever results match any single filter. Thus, it may be useful to create multiple filters for the same provider, each with a different single IID or a different set of multiple IIDs.

Note: You must initiate separate interest operations for all of the providers and interfaces in your query filters.

The signature for the method used to add filters in this guide is *DOFQueryBuilder_ AddFilter (DOFQueryBuilder, uint32, const DOFObjectID[], uint32, const DOFInterfaceID[])*.

Restrictions

Restrictions enable you to limit the results of a query to just those interfaces requested in a filter. For example, if you create a filter for a single IID, and you do not want to receive interface added calls for any additional interfaces that a provider provides, you could create a restriction for that IID. Restrictions are not used in our samples. Instead, we ignore non-matching results by checking the IID received in the callback against the IID we are interested in.

Sample Code

The following sample shows how to build a DOFQuery that looks for any providers of the Time-Based Alarm interface:

```
DOFInterfaceID iids[1];

iids[0] = TBAInterface_IID;
DOFQueryBuilder queryBuilder = DOFQueryBuilder_Create();
DOFQueryBuilder_AddFilter(queryBuilder, 0, NULL, 1, iids);
DOFQuery query = DOFQueryBuilder_BuildAndDestroy(queryBuilder);
```

Code Discussion

The *DOFQueryBuilder_AddFilter* function requires that the single IID be placed in an array. We pass NULL for the DOFObjectID array argument to query for any providers of the interface. The function also requires arguments for the count of the items in each array.

Both the DOFQueryBuilder and the DOFQuery must be destroyed. The DOFQueryBuilder is destroyed when the DOFQuery is created in the *DOFQuery_BuildAndDestroy* function. The DOFQuery will be destroyed in a later sample after we initiate the query operation.

Implementing a Query Listener

The DOFSystemQueryCallback interface provides the callbacks where you will receive the results of a query operation. The following callbacks are in this interface:

interfaceAdded. If a provider that matches the filter is discovered, this callback is called for every interface that the provider provides, whether or not the interface was included in the filter (unless you have also set restrictions). The common use for this callback is to create DOFObject instances for the found providers, as shown in the sample code.

interfaceRemoved. This callback is called if a found provider cancels one of its provide operations, but the provider still matches one of the filters in the query. While this callback can be used for advanced functionality, it is not covered in this guide. The sample passes NULL for this function when creating the function structure.

providerRemoved. This callback is called whenever a provider cancels one of its provide operations and this causes the provider to no longer match any of the query filters. The common use for this callback is to destroy DOFObject instances created in the *interfaceAdded* callback, as shown in the sample code.

complete. This callback is called when the query operation is canceled. The sample uses this callback to destroy the query operation.

Sample Code

The following shows an implementation of the DOFSystemQueryCallback interface that creates objects in the *interfaceAdded* callback and assigns them to predeclared variables, then destroys them in the *providerRemoved* callback and sets the variables back to NULL (note that with this code, you can only find two providers at a time):

```
static DOFObject provider = NULL;
static DOFObject provider2 = NULL;

static void queryComplete(DOFSystemQueryCallback self, DOFOperation operation,
         DOFException except);
static void interfaceAdded(DOFSystemQueryCallback self, DOFOperation operation,
         DOFObjectID objectID, DOFInterfaceID interfaceID);
static void providerRemoved(DOFSystemQueryCallback self, DOFOperation operation,
         DOFObjectID objectID);

static const struct DOFSystemQueryCallbackFns_t queryCallbackFns = {complete,
         interfaceAdded, NULL, providerRemoved};
static DOFSystemQueryCallback_t queryCallback = {&queryCallbackFns};

static void queryComplete(DOFSystemQueryCallback self, DOFOperation operation,
         DOFException except){
   DOFOperation_Destroy(operation);
}

static void interfaceAdded(DOFSystemQueryCallback self, DOFOperation operation,
         DOFObjectID objectID, DOFInterfaceID interfaceID){
   if(DOFInterfaceID_Compare(interfaceID, TBAInterface_IID) == 0){
```

```
        if(provider == NULL){
            provider = DOFSystem_CreateObject(system, objectID);
        }else if (provider2 == NULL){
            provider2 = DOFSystem_CreateObject(system, objectID);
        }
    }
}

static void providerRemoved(DOFSystemQueryCallback self, DOFOperation operation,
          DOFObjectID objectID){
    if(DOFObjectID_Compare(objectID, DOFObject_GetObjectID(provider)) == 0){
        DOFObject_Destroy(provider);
        provider = NULL;
    }

    if(DOFObjectID_Compare(objectID, DOFObject_GetObjectID
              (provider2)) == 0){
        DOFObject_Destroy(provider2);
        provider2 = NULL;
    }
}
```

Discovering Multiple Providers

With a query built and a query listener in place that can handle multiple providers, we are almost ready to initiate a query operation that uses them. However, before we can discover multiple providers, you need to express interest in multiple providers. The interest operation we showed before expressed interest in a single provider. If we used it in combination with our new query, the query would find only the single provider. (Queries can only find the results of interest operations.)

Expressing Interest

To express interest in multiple providers, you could call *DOFSystem_BeginInterest* again for each of those you are interested in. However, you could also pass a multicast address for the DOFObjectID when initiating an interest operation.

Sample Code

The library provides a broadcast address constant that we used to initiate an interest operation in the following sample:

```
DOFOperation interestOp = DOFSystem_BeginInterest(system, DOFOBJECTID_BROADCAST,
          TBAInterface_IID, DOFINTERESTLEVEL_WATCH, NULL, DOF_TIMEOUT_NEVER, NULL,
          NULL);
```

While it is possible to multicast in other ways, these are transport-dependent and not covered in this guide.

Initiating a Query Operation

The signatures for the method that initiates the query operation is *DOFSystem_BeginQuery (const DOFSystem, const DOFQuery, uint32, const DOFSystemQueryCallback, void *)*.

Sample Code

The following sample shows the operation (the DOFQuery instance created earlier in this chapter is passed to the *DOFSystem_BeginQuery* function):

```
DOFOperation queryOp;

...

queryOp = DOFSystem_BeginQuery(system, query, DOF_TIMEOUT_NEVER, &queryCallback,
          NULL);
DOFQuery_Destroy(query);

...

if(queryOp){
   DOFOperation_Cancel(queryOp);
}
```

Code Discussion

The following steps explain the sample code above:

1. Declare a DOFOperation.
   This operation needs to be used in multiple blocks.
2. Call the *DOFSystem_BeginQuery* function.
   Pass the DOFQuery instance and the operation listeners that were created earlier.
3. Cancel the query operation.
4. Destroy the query operation.
   This is not shown in the above sample. For more information, the complete call of the DOFSystemQueryCallback interface.

Chapter 6: Single-Transaction Asynchronous Operations

The single-transaction asynchronous operations mirror the functionality available in synchronous operations: you can get or set a DOF interface property value and invoke a DOF interface method. The main advantage of these operations for the requestor is that they are non-blocking. Another advantage is that it becomes possible to receive results from multiple providers, where synchronous operations return only the first response received.

Functionality on the provider does not change for these asynchronous operations: the same methods implemented for the corresponding synchronous operations respond to the requestor. Consequently, this chapter contains no sample code for the provider.

The samples in this chapter frequently use a DOFObject variable called *broadcastObject*. This is a broadcast DOFObject created using the broadcast DOFObjectID provided by the library.

Getting Properties Asynchronously

The following diagram shows the communication sequence for an asynchronous get operation:

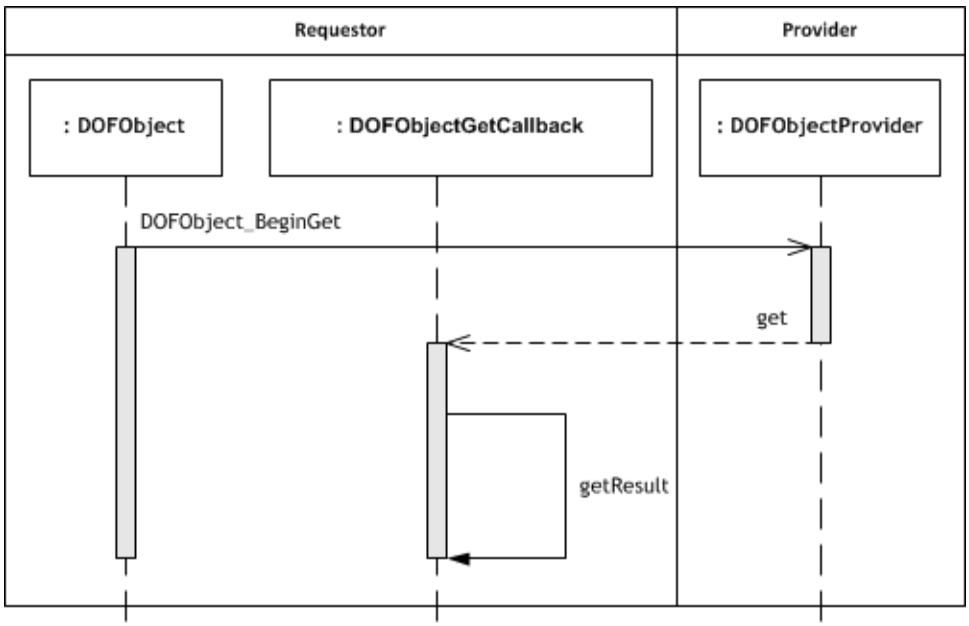

Figure 4. Asynchronous Get Operation Sequence Diagram

Get Operation Listener

As shown in the sequence diagram, asynchronous get operations require the requestor to implement the DOFObjectGetCallback interface to handle the results of the *DOFObject_BeginGet* call. This interface has the following callbacks:

getResult. This is called whenever the requestor receives the result of a get operation.

complete. This is called when the asynchronous get operation times out or is canceled. You can use it to clean up any application logic associated with the operation. It is also often a good practice to use this function to destroy the get operation.

Sample Code

The following sample shows an implementation of DOFObjectGetCallback that immediately prints received results to the console:

```
static void getComplete(DOFObjectGetCallback self, DOFOperation operation,
         DOFException except);
static void getResult(DOFObjectGetCallback self, DOFOperation operation,
         DOFProviderInfo providerInfo, DOFValue value, DOFException except);
static const struct DOFObjectGetCallbackFns_t getCallbackFns = {getComplete,
         getResult};
static DOFObjectGetCallback_t getCallback = {&getCallbackFns};

static void getComplete(DOFObjectGetCallback self, DOFOperation operation,
         DOFException except){
   DOFOperation_Destroy(operation);
}

static void getResult(DOFObjectGetCallback self, DOFOperation operation,
         DOFProviderInfo providerInfo, DOFValue value, DOFException except){
   boolean unwrappedResult;

   if(!except){
      unwrappedResult = DOFValueBoolean_Get(value);
      printf("Results of the begin get operation. Alarm Active Property = %s\n",
               (unwrappedResult ? "True": "False"));
   }

   if(except){
      /* Handle the error. */
   }
}
```

Code Discussion

Note that the library is entirely flexible in allowing you to handle the get results in any way you choose.

Calling *DOFObject_BeginGet*

The signature for this method is *DOFObject_BeginGet (DOFObject, const DOFInterfaceProperty, DOFOperationControl, uint32, const DOFObjectGetCallback, void *).*

Sample Code

The following sample shows this call:

```
void Requestor_sendBeginGetRequest(){
   DOFOperation op;
   DOFInterfaceProperty property DOFInterface_GetProperty (TBAInterface_DEF,
            TimeBasedAlarm_AlarmActiveID);
   DOFObject_BeginGet(broadcastObject, property, NULL, 10000, &getCallback,
            NULL);

   if(op == NULL){
      printf("The get operation failed.\n");
   }
}
```

Code Discussion

DOFObject_BeginGet returns a boolean if the call fails, so the sample also checks for the failure. The specific errors that caused the failure are passed to the *getResult* callback in the DOFObjectGetCallback interface.

Setting Properties Asynchronously

The following diagram shows the communication sequence for an asynchronous set operation.

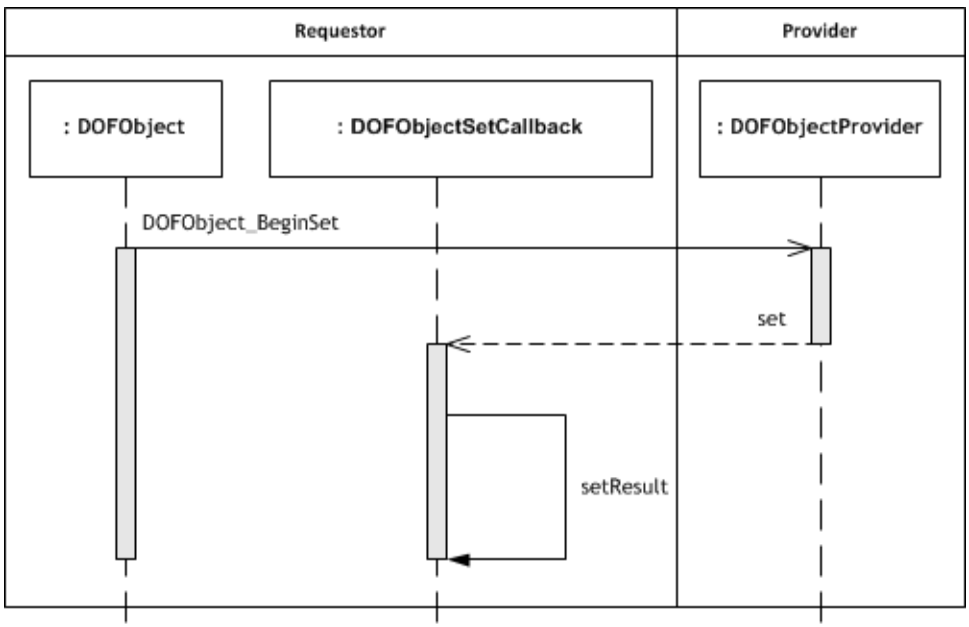

Figure 5. Asynchronous Set Operation Sequence Diagram

Set Operation Listener

Asynchronous set operations require the requestor to implement the DOFObjectSetCallback interface to handle the results of the *DOFObject_BeginSet* call. This interface has the following callbacks:

setResult. This is called whenever the requestor receives the result of a set operation.

complete. This is called when the asynchronous set operation times out or is canceled. You can use it to clean up any application logic associated with the operation. It is also often a good practice to use this function to destroy the set operation.

Sample Code

The following sample shows an implementation of DOFObjectSetCallback that immediately prints received results to the console:

```
static void setComplete(DOFObjectSetCallback self, DOFOperation operation,
         DOFException except);
static void setResult(DOFObjectSetCallback self, DOFOperation operation,
         DOFProviderInfo providerInfo, DOFException except);
static const struct DOFObjectSetCallbackFns_t setCallbackFns = {setComplete,
         setResult};
static DOFObjectSetCallback_t setCallback = {&setCallbackFns};

static void setComplete(DOFObjectSetCallback self, DOFOperation operation,
         DOFException except){
   DOFOperation_Destroy(operation);
}

static void setResult(DOFObjectSetCallback self, DOFOperation operation,
         DOFProviderInfo providerInfo, DOFException except){
   if(!except){
      printf("Successfully set the property for Provider %d.\n");
   }

   if(except){
      /* Handle the error. */
   }
}
```

Calling *DOFObject_BeginSet*

The signature for this method is *DOFObject_BeginSet (DOFObject, const DOFInterfaceProperty, DOFValue, DOFOperationControl, uint32, const DOFObjectSetCallback, void *).*

Sample Code

The following sample shows this call:

```
void Requestor_sendBeginSetRequest(boolean active){
   DOFOperation op;
   DOFInterfaceProperty property;
```

```
    DOFValue setValue = DOFValueBoolean_Create(active);
    property = DOFInterface_GetProperty(TBAInterface_DEF,
             TimeBasedAlarm_AlarmActiveID);
    op = DOFObject_BeginSet(broadcastObject, property, setValue, NULL, 5000,
             &setCallback, NULL);

    DOFValue_Destroy(setValue);

    if(op == NULL){
       printf("The Set operation failed.\n");
    }
}
```

Invoking Methods Asynchronously

The following diagram shows the communication sequence for an asynchronous invoke operation.

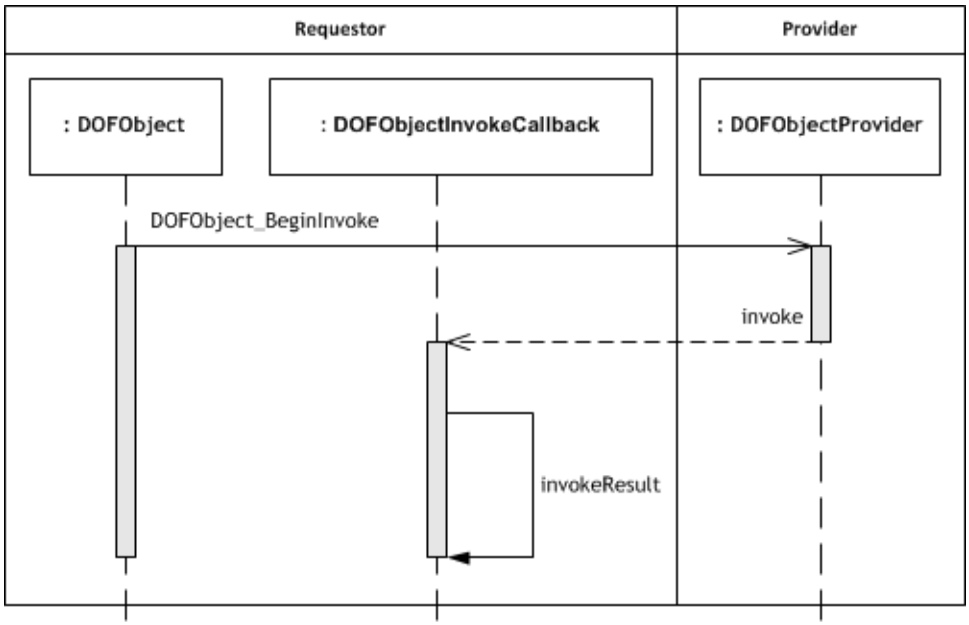

Figure 6. Asynchronous Invoke Operation Sequence Diagram

Invoke Operation Listener

Asynchronous invoke operations require the requestor to implement the DOFObjectInvokeCallback interface to handle the results of the *DOFObject_BeginInvoke* call. This interface has the following callbacks:

invokeResult. This is called whenever the requestor receives the result of an invoke operation.

complete. This is called when the asynchronous invoke operation times out or is canceled. You can use it to clean up any application logic associated with the operation. It is also often a good practice to use this function to destroy the invoke operation.

Sample Code

The following sample shows an implementation of DOFObjectInvokeCallback that immediately prints received results to the console:

```
static void invokeComplete(DOFObjectInvokeCallback self, DOFOperation operation,
         DOFException except);
static void invokeResult(DOFObjectInvokeCallback self, DOFOperation operation,
         DOFProviderInfo providerInfo, uint16 resultCount,
         const DOFValue result[], DOFException except);
static const struct DOFObjectInvokeCallbackFns_t invokeCallbackFns =
         {invokeComplete, invokeResult};
static DOFObjectInvokeCallback_t invokeCallback = {&invokeCallbackFns};

static void invokeComplete(DOFObjectInvokeCallback self, DOFOperation operation,
         DOFException except){
   DOFOperation_Destroy(operation);
}

static void invokeResult(DOFObjectInvokeCallback self, DOFOperation operation,
         DOFProviderInfo providerInfo, uint16 resultCount,
         const DOFValue result[], DOFException except){
   boolean unwrappedResult;
   if(!except){
      unwrappedResult = DOFValueBoolean_Get(result[0]);
      printf("Results of the invoke operation from provider %d. Alarm Active
               Property = %s\n", (unwrappedResult ? "True": "False"));
   }else{
      /* Handle the error. */
   }
}
```

Calling *beginInvoke DOFObject_BeginInvoke*

The signatures for this method is *DOFObject_BeginInvoke (DOFObject, const DOFInterfaceMethod, uint16, const DOFValue parameters[], DOFOperationControl, uint32, const DOFObjectInvokeCallback, void *)*. As with similar functions, the uint16 parameter is the count of items in the DOFValue array.

Sample Code

The following sample shows this call:

```
void Requestor_sendASynchInvokeRequest(DateTime alarmTime){
   DOFOperation op;
   DOFInterfaceMethod method;

   DOFValue inputParameter = DOFValueDateTime_Create(alarmTime);
   DOFValue parameterArray[1];

   parameterArray[0] = inputParameter;
   method = DOFInterface_GetMethod(TBAInterface_DEF,
            TimeBasedAlarm_SetNewTimeID);
   op = DOFObject_BeginInvoke(broadcastObject, method, 1, parameterArray, NULL,
            5000, &invokeCallback, NULL );

   if(op == NULL){
```

```
        printf("The Invoke operation failed.\n");
    }

    DOFValue_Destroy(inputParameter);
}
```

Chapter 7: Ongoing Asynchronous Operations

In this chapter, we'll discuss two ongoing asynchronous operations that deal directly with items of interface functionality:

1. Subscribing to a DOF interface property
2. Registering for a DOF interface event

You will learn how to implement these operations on the provider, how to implement operation listeners, and how to initiate the operation with the requestor.

Subscribing to a Property

The following diagram shows the communication sequence for a subscribe operation.

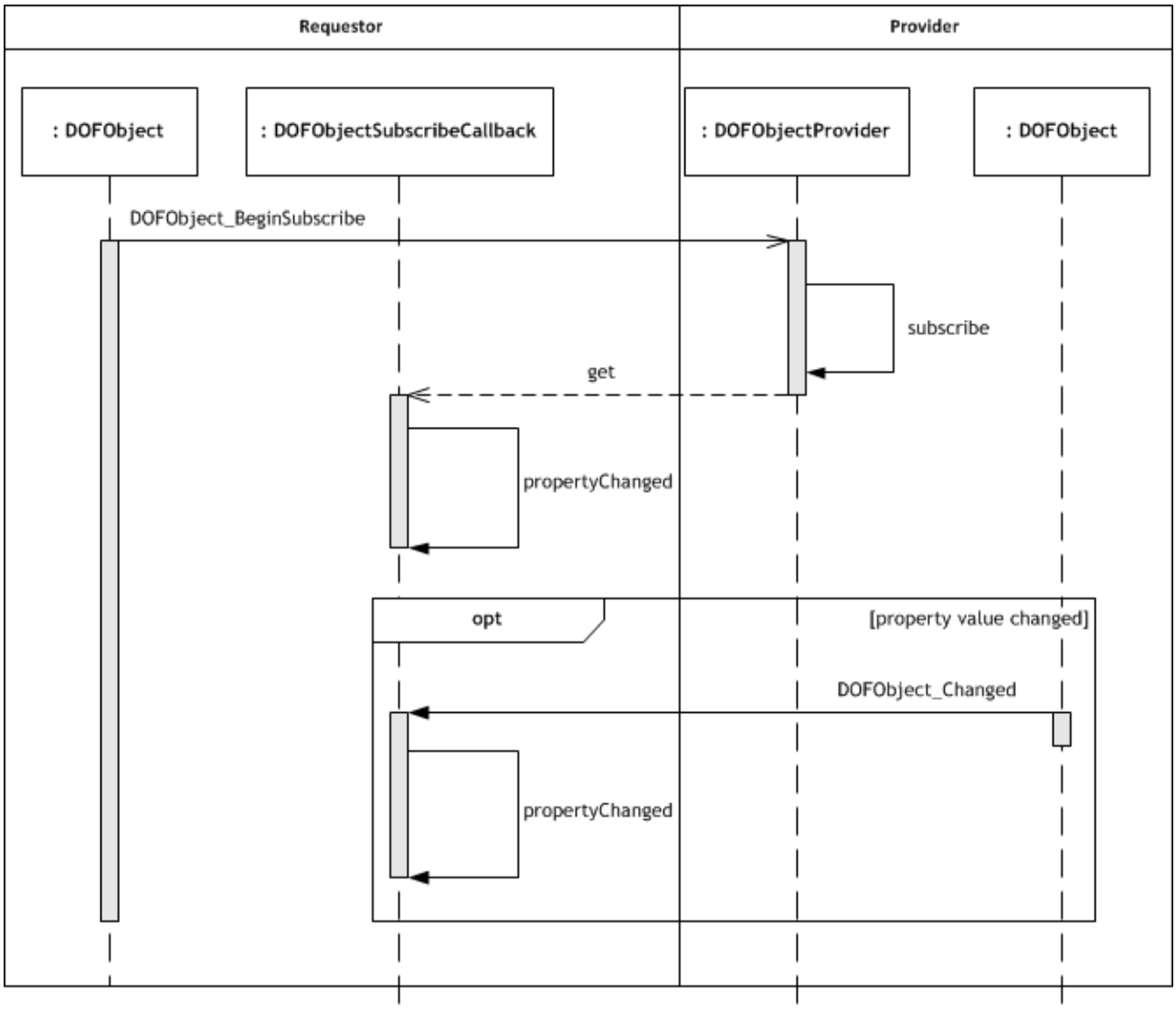

Figure 7. Subscribe Operation Sequence Diagram

Providers

If *all* the following conditions are true, the provider does not need to do any additional work to implement subscribe operations:

- You have already implemented the *get* callback for the property.
  As you can see from the sequence diagram, it is the *get* callback that sends the requestor the initial value of the property.
- You have not implemented the *subscribe* callback and are passing NULL for it in the function structure.
  The default behavior is to automatically accept subscribe requests.
  Note: The *subscribe* method is for the provider's internal use. For example, you could use it if you want the provider to track the requestors that are subscribed to a property. If you do override this method, you must ensure that you call *DOFRequest_Return* or *DOFRequest_Throw*. Using *DOFRequest_Throw* causes the requestor's subscription to fail. Samples of the provider's *subscribe* callback are not shown in this guide.
- There is no way for the property value to change other than through set operations from requestors.
  The library automatically notifies subscribed requestors if the property value changes through a set operation. If, however, the property can be changed by the *invoke* callback or you have internal application logic that can cause a property value to change, you must call the *changed* method.

Sample Code

The following sample shows the DOFObject_Changed method:

```
static void setActive(int providerIndex, boolean active)
   if(isActive != active){
      isActive = active;
      DOFInterfaceProperty property = DOFInterface_GetProperty (TBAInterface_DEF,
               TimeBasedAlarm_AlarmActiveID)
      DOFObject_Changed(provider, property);
   }
}
```

Subscribe Operation Listener

Subscribe operations require the requestor to implement the DOFObjectSubscribeCallback interface to handle the results of the *DOFObject_BeginSubscribe* call. This interface has the following methods:

propertyChanged. This is called whenever the provider sends notification that a property value has changed. It is also called immediately after initiating the operation when the provider sends the current property value. Finally, it is called with an exception if the provider rejects the subscribe request.

complete. This is called when the subscribe operation is canceled or the subscription fails due to an error (such as a timeout). You can use it to clean up any application logic associated with the operation. It is also often a good practice to use this function to destroy the subscribe operation.

Sample Code

The following sample shows an implementation of DOFObjectSubscribeCallback that prints received results to the console:

```
static void subscribeComplete(DOFObjectSubscribeCallback self,
         DOFOperation operation, DOFException except);
static void subscribePropertyChanged(DOFObjectSubscribeCallback self,
         DOFOperation operation, DOFProviderInfo providerInfo, DOFValue value,
         DOFException except);
static const struct DOFObjectSubscribeCallbackFns_t subscribeCallbackFns =
         {subscribeComplete, subscribePropertyChanged};
static DOFObjectSubscribeCallback_t subscribeCallback = {&subscribeCallbackFns};

static void subscribeComplete(DOFObjectSubscribeCallback self,
         DOFOperation operation, DOFException except){
   DOFOperation_Destroy(operation);
}

static void subscribePropertyChanged(DOFObjectSubscribeCallback self,
         DOFOperation operation, DOFProviderInfo providerInfo, DOFValue value,
         DOFException except){
   boolean unwrappedResult;

   if(!except){
      unwrappedResult = DOFValueBoolean_Get(value);
      printf("Received changed value from provider %d. Alarm Active Property =
               %s\n", (unwrappedResult ? "True": "False"));
   }else{
      /* Handle errors. */
   }
}
```

Calling *DOFObject_BeginSubscribe*

The signature for this method is *DOFObject_BeginSubscribe (DOFObject, const DOFInterfaceProperty, uint32, DOFOperationControl, uint32, const DOFObjectSubscribeCallback, void *).*

Sample Code

The following sample shows the *DOFObject_BeginSubscribe* call:

```
void Requestor_sendBeginSubscribeRequest(DOFObject provider){
   DOFOperation op;
   DOFInterfaceProperty property = DOFInterface_GetProperty (TBAInterface_DEF,
            TimeBasedAlarm_AlarmActiveID);

   op = DOFObject_BeginSubscribe(provider, property, 0, NULL, DOF_TIMEOUT_NEVER,
            &subscribeCallback, NULL);

   if(op == NULL){
```

```
        printf("Subscribe operation failed.");
    }
}
```

Code Discussion

A couple of parameters deserve consideration:

First uint32. This is the minimum period parameter. It represents the interval that you want between property changed notifications for the same provider and property. This means that if the property changes more frequently, you will be notified of the most recent change only after the minimum period has expired.

Second uint32. This is a timeout parameter. While the timeout for other operations represented the maximum amount of time in milliseconds that you are willing to wait for results, the timeout in ongoing operations represents the amount of time in milliseconds that you want the operation to *persist*. It is for this reason that the sample indicates that the operation should never time out. When using the TIMEOUT_NEVER constant, you should ensure that you cancel the operation.

Registering for an Event

The following diagram shows the communication sequence for a register operation.

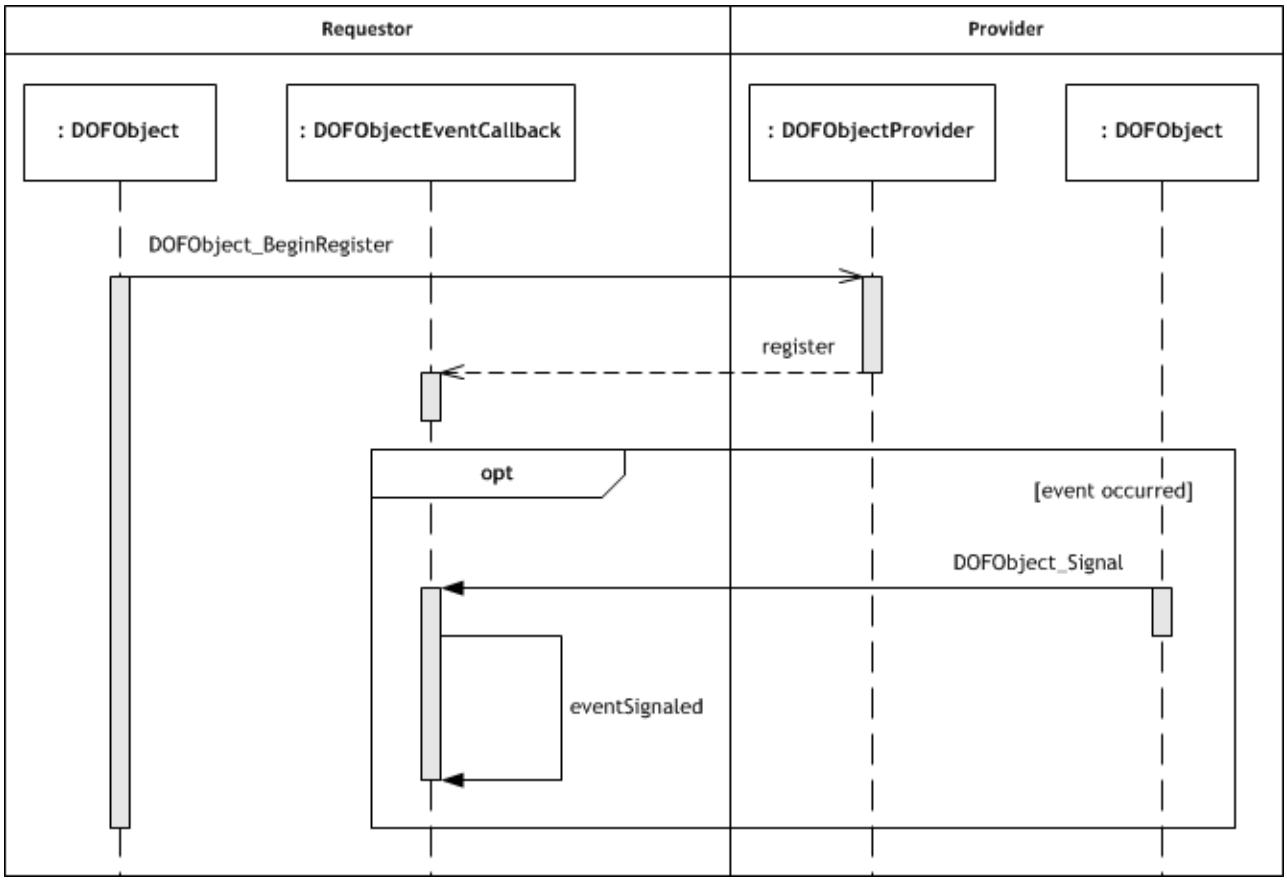

Figure 8. Register Operation Sequence Diagram

Providers

Implementing the *register* method is optional for the provider. Like the *subscribe* method, it is for the provider's internal use. If you do implement it, the same rules apply for calling *DOFRequest_Return* or *DOFRequest_Throw* that apply to the *subscribe* method.

However, unlike the *DOFObject_Changed* method for subscriptions, which the provider need only call under specific conditions, the *DOFObject_Signal* method must always be called to enable event notification.

Sample Code

The following sample shows the *DOFObject_Signal* method:

```
static void checkForEvent(void){
   if(isActive){
      if(alarmTime - getCurrentTime() <= 0){
         DOFInterfaceEvent event = DOFInterface_GetEvent
                  (TBAInterface_DEF, TimeBasedAlarm_AlarmTriggeredID);
         if(DOFObject_IsSignalRequired(provider, event)){
            DOFObject_Signal(provider, event, 0, NULL);
            alarmTime = 0;
         }
      }
   }
}
```

Code Discussion

Follow these steps for using the *DOFObject_Signal* function:

1. Check for the conditions that should trigger the event.
2. Call *DOFObject_IsSignalRequired* to determine whether any requestors are registered for the event.
3. Package the output parameters.
4. Call the *DOFObject_Signal* function.

Register Operation Listener

Register operations require the requestor to implement the DOFObjectEventCallback interface to handle the results of the *DOFObject_BeginRegister* call. This interface has the following methods:

eventSignaled. This is called when an event is triggered. It is also called immediately after initiating the operation, when the provider sends an initial acknowledgment. Finally, it is called with an exception if the provider rejects the register request.

complete. This is called when the register operation is canceled or the registration fails due to an error (such as a timeout). You can use it to clean up any application logic associated with the operation. It is also often a good practice to use this function to destroy the register operation.

Sample Code

The following sample shows an implementation of DOFObjectEventCallback that prints received results to the console:

```
static void eventComplete(DOFObjectEventCallback self, DOFOperation operation,
         DOFException except);
static void eventSignaled(DOFObjectEventCallback self, DOFOperation operation,
         DOFProviderInfo providerInfo, uint16 parameterCount,
         const DOFValue parameters[], DOFException except);
static const struct DOFObjectEventCallbackFns_t eventCallbackFns =
         {eventComplete, eventTriggered};
static DOFObjectEventCallback_t eventCallback = {&eventCallbackFns};

static void eventComplete(DOFObjectEventCallback self, DOFOperation operation,
         DOFException except){
   DOFOperation_Destroy(operation);
}

static void eventSignaled(DOFObjectEventCallback self, DOFOperation operation,
         DOFProviderInfo providerInfo, uint16 parameterCount,
         const DOFValue parameters[], DOFException except){
   if(!except){
      if(parameters == NULL){
         printf("Received ack from provider %d.\n");
      }else{
         printf("Received event from provider %d.\n");
      }
   }else{
      /* Handle errors. */
   }
}
```

Code Discussion

The *eventSignaled* callback is initially called when the provider acknowledges the registration, which does not mean the event was triggered. It is important that code in this callback distinguish between the initial acknowledgement and subsequent calls when the event has been triggered:

- When an event was triggered, the DOFValue array argument will always be filled. If the event has no output parameters, the array will be empty, but not null.
- For an acknowledgement, the array argument will be null. The DOFException argument will also be null.
- If the provider rejects the registration, the DOFException argument will be filled and the DOFValue array argument will be null.

Calling *DOFObject_BeginRegister*

The signature for this method is *DOFObject_BeginRegister (DOFObject, const DOFInterfaceEvent, DOFOperationControl, uint32, const DOFObjectEventCallback, void *).*

Sample Code

The following sample shows the *DOFObject_BeginRegister* call:

```
void Requestor_sendRegister(DOFObject provider){
   DOFOperation op;

   DOFInterfaceEvent event = DOFInterface_GetEvent(TBAInterface_DEF,
            TimeBasedAlarm_AlarmTriggeredID);

   op = DOFObject_BeginRegister(provider, event, NULL, DOF_TIMEOUT_NEVER,
            &eventCallback, NULL);

   if(op == NULL){
      printf("Register operation failed.");
   }
}
```

Chapter 8: Activate-Level Interest

Earlier, we discussed how a requestor can perform interest and query operations to discover providers, and we introduced the concept of activate-level interest. To review, activate-level interest is a request to begin providing an interface. Up to this point in the guide, our provider has performed a provide operation when it initializes. This chapter covers how to program a provider to begin dynamically providing an interface in response to activate-level interest.

To respond to activate-level interest, the provider must implement the DOFSystemActivateInterestListener interface. The library then notifies the system when an activate- or higher level interest is received. This interface has the following methods:

removed. The library calls this function when the listener is removed or the system is destroyed. You can use it to free resources associated with the listener; however, since our sample code does not allocate any resources for the listener, we pass NULL in the function pointer struct to ignore removal of the listener.

activate. The library calls this method when an activate-level interest is received.

cancelActivate. The library calls this method when an interest request is canceled or times out.

This chapter will show how to implement the interface. After implementing it, you would add it to a DOFSystem instance using the *DOFSystem_AddActivateInterestListener* function.

Implementing DOFSystemActivateInterestListener

Implementing DOFSystemActivateInterestListener is similar to implementing the other listeners shown in this guide.

Sample Code

The following sample shows an implementation of the interface (functions will be shown later in this chapter):

```
static void ActivateInterestListener_Activate
         (DOFSystemActivateInterestListener self, DOFSystem system,
         DOFRequest request, DOFObjectID objectID, DOFInterfaceID interfaceID);
static void ActivateInterestListener_CancelActivate
         (DOFSystemActivateInterestListener self, DOFSystem system,
         DOFRequest request, DOFObjectID objectID, DOFInterfaceID interfaceID);

static const struct DOFSystemActivateInterestListenerFns_t
         activateInterestListenerFns = {NULL, ActivateInterestListener_Activate,
         ActivateInterestListener_CancelActivate};

DOFSystemActivateInterestListener_t activateInterestListener =
         {&activateInterestListenerFns};
```

Code Discussion

Because we are not planning to use the removed function, we pass NULL for it when we declare the struct of function pointers.

Writing the *activate* Method

Use the *activate* method of the DOFSystemActivateInterestListener interface to determine whether the interest applies to your provider and to begin providing the interface if it does.

Sample Code

The following sample shows an implementation of the *activate* method:

```
void ActivateInterestListener_Activate (DOFSystemActivateInterestListener self,
         DOFSystem system, DOFRequest request, DOFObjectID objectID,
         DOFInterfaceID interfaceID){
   if(DOFObjectID_Compare(objectID, providerID) == 0 || DOFObjectID_Compare
            (objectID, DOFOBJECTID_BROADCAST == 0){
      if(DOFInterfaceID_Compare(interfaceID,
               TimeBasedAlarmInterface_IID) == 0){
         interestedCount++;
         if(!provider){
            provider = DOFSystem_CreateObject(system, objectID)
            provideOperation = DOFObject_BeginProvide(provider,
                     TimeBasedAlarmInterface_DEF, DOF_TIMEOUT_NEVER, &provider,
                     NULL);
         }
      }
   }
}
```

Code Discussion

The sample code does the following:

- Examines the OID in the interest to determine whether the interest applies to the provider. If the OID is the broadcast OID or the provider's specific OID, the interest can be assumed to apply to the provider.
- Examines the IID in the interest. If the OID is the broadcast OID, this step further helps to determine whether the interest applies to the provider. In addition, if the provider were capable of providing multiple interfaces, this step would be vital to determining which interface to provide in response to the interest.
- Keeps and increments a count of interest operations received for the interface. This is important so that the provider does not stop providing an interface if multiple requestors have expressed interest. We'll discuss this further in the next section.
- Creates a DOFObject. This code assumes that the DOFObject was declared and set to null in a global variable with the variable name *provider*. The DOFObject instance is then dynamically created in response to the interest. This is a reasonable use case, but you might also wish to create DOFObject instances elsewhere.

- Checks to see whether a provide operation (a previously declared DOFOperation) for the interface is null. If not, it begins providing the interface.

Writing the *cancelActivate* Method

Use the *cancelActivate* method of the DOFSystemActivateInterestListener interface to be notified when interest is canceled and stop providing.

Sample Code

The following sample shows an implementation of the *cancelActivate* method:

```
void ActivateInterestListener_CancelActivate
         (DOFSystemActivateInterestListener self, DOFSystem system,
         DOFRequest request, DOFObjectID objectID, DOFInterfaceID interfaceID){
   if(DOFObjectID_Compare(objectID, providerID) == 0 || DOFObjectID_Compare
            (objectID, DOFOBJECTID_BROADCAST) == 0){
      if(DOFInterfaceID_Compare(interfaceID,
               TimeBasedAlarmInterface_IID) == 0){
         interestedCount--;
         if(interestedCount <= 0){
            DOFOperation_Cancel(provideOperation);
            DOFOperation_Destroy(provideOperation);
            DOFObject_Destroy(provider);
            provider = NULL;
         }
      }
   }
}
```

Code Discussion

Like the *activate* method samples, this sample compares the OID and IID received to a known OID and IID to determine the binding the interest applies to. Notice that it decrements the interest count and does not stop providing until the count is zero or less.

Appendix A: Formatting OIDs

OIDs consist of both an OID class and data. The OID class must be a valid DOF OID class that has been formally registered, and the data is a number or string appropriate for identifying data in that class.

OIDs must be globally unique, and, in a production environment, they should be carefully managed. However, this appendix covers how you can create OIDs, using three classes that have already been registered, to use in experimentation and testing:

- GUID
- Domain
- Email

Note: All of these classes create large OIDs and may not be appropriate for resource-constrained devices. In addition, to create domain- or email-class OIDs, even for testing, you must own a properly registered Internet domain that meets Internet Engineering Task Force (IETF) specifications.

GUID Class OIDs

To create GUIDs that you may use as the data portion in OIDs requires only a tool that can randomly generate a 16-byte GUID in hexadecimal. Free browser-based GUID generators are readily available online.

Of the options outlined in this appendix, the GUID class has the advantage of being potentially the shortest type of OID as well as being freely available (not requiring that you own an Internet domain). Its potential disadvantage is that GUIDs are not as readily human-readable as domain names and email addresses, so that, in experimentation and testing, it may be easier to keep track of the objects being identified in your code when you use the other classes discussed in this appendix.

Domain Class OIDs

With your own registered Internet domain, you can create a very large number of domain class OIDs. The requirement for domain registry is to ensure that OIDs remain globally unique, although the OIDs do not actually use DNS resolution. For this reason, once you have a registered domain name, you can create multiple OIDs using any subdomain or other format that conforms to the IETF's RFC 2822, whether or not the resulting URL can actually be resolved by a DNS server. The only requirement is that you take responsibility for tracking the OID's uniqueness within your own domain. So, for example, if you owned "domain.com," you could use any of the following as OID data, even if you have not set up DNS resolution for them:

- domain.com
- requestor.domain.com
- provider.domain.com

As long as you own the domain and you manage unique subdomain names, the number of OIDs you can create is limited only by the maximum length of 63 bytes, the characters allowed by RFC 2822, and the portion of your OID data that is the fixed name of your registered domain.

Email Class OIDs

The email class extends the number of OIDs you can create using the same registered domain used in your domain class OIDs. You must format emails according to specifications outlined in RFC 2822.

As with the domain class, you may use subdomains that are not resolved by a DNS server. This provides options for managing uniqueness. You can use email classes with OIDs where the text string is different, but the domain is the same, as in the following examples:

- requestor@domain.com
- provider@domain.com

You can also create OIDs that share the same text string, but where subdomains are different, as in the following examples:

- object1@requestor.domain.com
- object1@provider.domain.com

In addition, for experimental and testing purposes only, it is reasonably safe to use your own email address or addresses as OIDs, even if you don't own their associated domains.

Note: The maximum length of an email class OID is 63 bytes.

Formatting OIDs for DOFObjectID Create Methods

After choosing the data you want to use in your OIDs, you need to know how to format the data to pass it into a DOFObjectID create method.

In the C OAL, it is generally easier to create GUID class OIDs using the *DOFObjectID_Create(DOFObjectIDClass, uint32, const uint8 *)* function, while it is usually easier to create domain and email class OIDs using the *DOFObjectID_Create_String(DOFObjectIDClass, const char *)* function.

The uint32 parameter in the *DOFObjectID_Create* function represents the size of the array passed in the const uint8 * parameter. For GUIDs, this must always be sixteen, so in the following sample we defined this as a constant. You must manually format the hexadeximal of the GUID into individual bytes to create an array.

Code Samples

The following is an example of how to create a GUID class DOFObjectID:

```
#define GUID_SIZE 16
```

```
uint8 guid[GUID_SIZE] = {0x05,0x02,0x34,0x70,0x6b,0xb1,0x45,0x32,
        0x89,0xd9,0x46,0x10,0x55,0xce,0xf6,0x0c};
DOFObjectID providerID = DOFObjectID_Create(DOFOBJECTIDCLASS_GUID, GUID_SIZE,
        guid);
```

The following is an example of how to create a domain class DOFObjectID:

```
DOFObjectID providerID = DOFObjectID_Create_String(DOFOBJECTIDCLASS_DOMAIN,
        "provider.opendof.org");
```

The following is an example of how to create an email class DOFObjectID:

```
DOFObjectID providerID = DOFObjectID_Create_String(DOFOBJECTIDCLASS_EMAIL,
        "provider@opendof.org");
```

Summary

The GUID, domain, and email OID classes provide nearly limitless possibilities for creating unique OIDs, which you can use freely in experimentation and testing. However, remember that in a production environment, it is vital to manage and monitor the uniqueness of OIDs. In addition, every OID class has its own specifications for use. System designers should carefully plan which OIDs will be used in a production environment.