SunXTL 1.1 Provider Programmer's Guide

The Network Is the Computer™

Sun Microsystems Computer Company
2550 Garcia Avenue
Mountain View, CA 94043 USA
415 960-1300 fax 415 969-9131

Part No.: 801-7049-11
Revision A, December 1995

Contents

Preface xi

1. Introduction to SunXTL Providers 1

Role of Providers in SunXTL Teleservices 1

The `XtlP` Classes 4

2. Provider Framework and Concepts 7

The Provider Framework 7

The Provider Programming Model 8

Requests and Event Indications 9

Relationships Between `XtlP` Classes 10

`XtlP` Classes—Factories, Providers, and Calls 11

Setting and Getting State Slots 11

Receiving Requests From the Client 11

Sending Indications to the Client 12

Call Status and Transitions 12

Interfacing to the Underlying Telephone Technology 18

Starting Your Provider Process 19
Creating Providers. 20
Performing Call Operations 22
Making an Outgoing Call 22
Creating a Call 23
Starting an Outgoing Call 24
Monitoring Call Progress 25
Receiving Incoming Calls 26
Disconnecting Calls 28
Putting Calls On and Taking Calls Off Hold 30
Transferring a Call 31
Conferencing a Call 32
Guaranteeing Conference Indications 33
Dropping a Call From a Conference 37
Configuring Media Channels 38
Presenting Media Channels to the Client 39
3. Creating a Provider Package 43
Writing a Provider Program 43
Quick Start to Writing a Provider 43
Writing a Provider from the Beginning. 44
Linking to the SunXTL Libraries 46
Compiling the Provider Simulator 46
Creating a Provider Package 46
Creating Relocatable Packages. 47

Creating Provider Templates 47
Creating Template Files at Installation. 49
Template File Format. 49
4. Utility Classes 53
Using `XtlByteArray`. 53
Using `XtlString` 55
Using `XtlKVList` 57
Copying an `XtlKVList` 62
Creating a Hierarchical `XtlKVList` 62
Traversing `XtlKVLists` 63
Using the Event Dispatcher 64
Initializing the Dispatcher 66
Using `dpIOHandler`. 68
Using the Database Query Functions 69
Using `XtlFormat` 70
Requesting Formats. 73
Usage Examples. 73
Using `XtlCallReference` 75
5. `XtlP` Classes 77
`XtlP` Base Class 78
Exception Codes. 78
Error Codes. 79
Cause Codes 80
State Slots 81

`XtlPFactory` Methods 83
`XtlPProvider` Methods 85
`XtlPCall` Methods 88
Call Event and Request Enumerations 94
`XtlPPort` Methods 96
Configuring Media Channels With `configuration_req()` . 96
Handling Media Channel Configurations 97
Some Typical Configuration Pairs 98
Index 101

Figures

Figure 1-1 XTL Platform Architecture 2
Figure 1-2 Provider Programming Framework 3
Figure 1-3 `XtlP` Class Hierarchy 4
Figure 2-1 Sources of Stimuli to a Provider 9
Figure 2-2 Provider Object Instantiation Relationships 10
Figure 2-3 Normal Call Status Transitions 14
Figure 2-4 Outgoing Call Scenario 23
Figure 2-5 Incoming Call Scenario 26
Figure 2-6 Call Disconnection Scenario 28
Figure 2-7 Call Hold and Unhold Scenario 30
Figure 2-8 Call Conference Scenario 33
Figure 2-9 Conference Call Scenario With Handle Swapping 35
Figure 2-10 Drop Call Scenario 38
Figure 4-1 `XtlKVList` Structure 58
Figure 4-2 `dpDispatcher` and `dpIOHandler` Interactions 64
Figure 4-3 Voice Format Specification Example 71

Figure 5-1 `XtlP` Class Hierarchy 77

Tables

Table 1-1 Additional MPI Classes and Data Types 5

Table 2-1 Call Status Enumerations 15

Table 2-2 Valid Requests for Call Status 17

Table 2-3 Valid Indication Events for Call Status Transitions 18

Table 3-1 Generic Provider Code Files 44

Table 3-2 Library Names and Functions 46

Table 3-3 Key-Value Pairs for Provider Template Files 48

Table 4-1 `XtlByteArray` Methods (from `bytearray.h`) 54

Table 4-2 `XtlString` Methods (from `bytearray.h`) 55

Table 4-3 `XtlKVList` Methods (from `kvlist.h`) 59

Table 4-4 `dpDispatcher` Methods (from `dispatcher.h`) 65

Table 4-5 `dpIOHandler` Methods (from `iohandler.h`) 68

Table 4-6 Database Query Functions (from `xtldb.h`) 69

Table 4-7 Predefined `XtlFormat` Keys and Values 72

Table 5-1 Exception Codes (from `xtlp_globals.h`) 78

Table 5-2 Error Codes (from `xtlp_globals.h`) 79

Table 5-3 Cause Codes (from `xtlp_globals.h`) 80
Table 5-4 State Slots for `XtlP` Classes 82
Table 5-5 `XtlPFactory` Class Methods (from `xtlp_factory.h`) ... 83
Table 5-6 `XtlPFactory` Request and Event Codes (from `xtlp_factory.h`)..... 84
Table 5-7 `XtlPProvider` Interface Methods (from `xtlp_provider.h`) 85
Table 5-8 `XtlPProvider` Request and Event Codes (from `xtlp_provider.h`)..... 87
Table 5-9 `XtlPCall` Class Methods (from `xtlp_call.h`) 88
Table 5-10 Call Event Codes (from `xtlp_call.h`) 94
Table 5-11 XtlPCall Class Request Enumerations (from `xtlp_call.h`) 95
Table 5-12 Configuration Pairs for `configuration_req()`..... 98

Preface

This book documents the SunXTL media platform interface (MPI) library and describes how to create an SunXTL provider package for end users. The MPI is one of two interfaces provided by the SunXTL Teleservices for the Solaris™ platform; the other is the application programming interface (API), which is described in the *Sun XTL 1.1 Application Programmer's Guide.*

Note – The term telephone device is used in this manual. However, SunXTL Teleservices can be used with any teleservices device. If you are not using a telephone device, replace the term telephone device with teleservices device.

The MPI enables you to write an SunXTL provider for your telephone device. By using your SunXTL provider, SunXTL applications can access the teleservices provided by your telephone device. A telephone device can be either hardware, software, or some combination thereof, that is capable of communicating with a telephone network.

Note that all references to Solaris in this book apply only to the SPARC version of Solaris.

Who Should Use This Book

This book helps you to write and package SunXTL providers that enable your telephone device (either hardware or software) to act as the teleservices technology of choice for SunXTL applications.

This book is intended for Solaris programmers who are knowledgeable in C++ and have an understanding of the SunXTL framework, which is described in the *Sun XTL 1.1 Architecture Guide*. This manual also assumes you have a thorough understanding of the telephone device for which you are writing an SunXTL provider.

How This Book Is Organized

Chapter 1, "Introduction to SunXTL Providers," introduces the concept of providers and the role providers play in the SunXTL architecture. An overview of the media platform interface is also given.

Chapter 2, "Provider Framework and Concepts," presents a high-level description of the provider framework and how the various MPI objects interact with each other.

Chapter 3, "Creating a Provider Package," describes the general structure of a provider program and the process of creating a provider package, including binaries, templates, and provider-specific documentation.

Chapter 4, "Utility Classes," describes the supporting classes, or utilities, that help tie the larger pieces together, such as global data types, data structures, and event handlers.

Chapter 5, "XtlP Classes," provides reference material for the interface methods of each MPI class.

Related Books

The following books are part of the SunXTL documentation set:

- *Sun XTL 1.1 Architecture Guide*
- *Sun XTL 1.1 Administrator's Guide*
- *Sun XTL 1.1 Application Programmer's Guide*
- *Sun XTL 1.1 Remote Client Mgr Guide*

What Typographic Changes and Symbols Mean

Table P-1 describes the type changes and symbols used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	**Meaning**	**Example**
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `system% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`system% su` `Password::`
	Also used to highlight class methods in tables	`virtual XtelPProvider();`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*`.`
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

Code samples are included in boxes and may display the following:

`%`	UNIX C shell prompt	`system%`
`$`	UNIX Bourne and Korn shell prompt	`system$`
`#`	Superuser prompt, all shells	`system#`
`Highlight`	A highlighted table row means the method must be overriden with your implementation code.	`virtual void cleanup()=0;`

Introduction to SunXTL Providers 1

The SunXTL media platform interface (MPI) is a standard programming interface for writing SunXTL *providers*. The MPI is implemented as a library called XTLP. This chapter describes the use and role of providers in the SunXTL architecture and introduces the classes in the XTLP library.

Role of Providers in SunXTL Teleservices

An SunXTL provider is a software entity that enables your telephone device to communicate with the SunXTL platform and provides SunXTL applications (herein referred to as *clients*) access to the services of your telephone device. And for your provider, the MPI is its interface to the SunXTL platform. In writing your provider, you only need to understand the MPI—in-depth understanding of the API and clients, while useful, is not required.

Providers allow SunXTL clients to set up, control, and terminate phone calls in a consistent manner. Providers also give clients access to the data associated with a call in a uniform manner. Additionally, value-added capabilities available from your telephone device are accessible to SunXTL clients through a simple, but powerful extension mechanism. These features and other important concepts are presented in Chapter 2, “Provider Framework and Concepts.”

Figure 1-1 shows the overall SunXTL architecture. A provider has two entry points (interfaces) to the SunXTL platform—the MPI and the SunXTL database interface (see “Using the Database Query Functions” on page 69). Consequently,

as a minimum, you do not need to be concerned with the other interfaces in the SunXTL platform. The only other assumption is that you have a thorough understanding of the telephone device for which you are writing a provider.

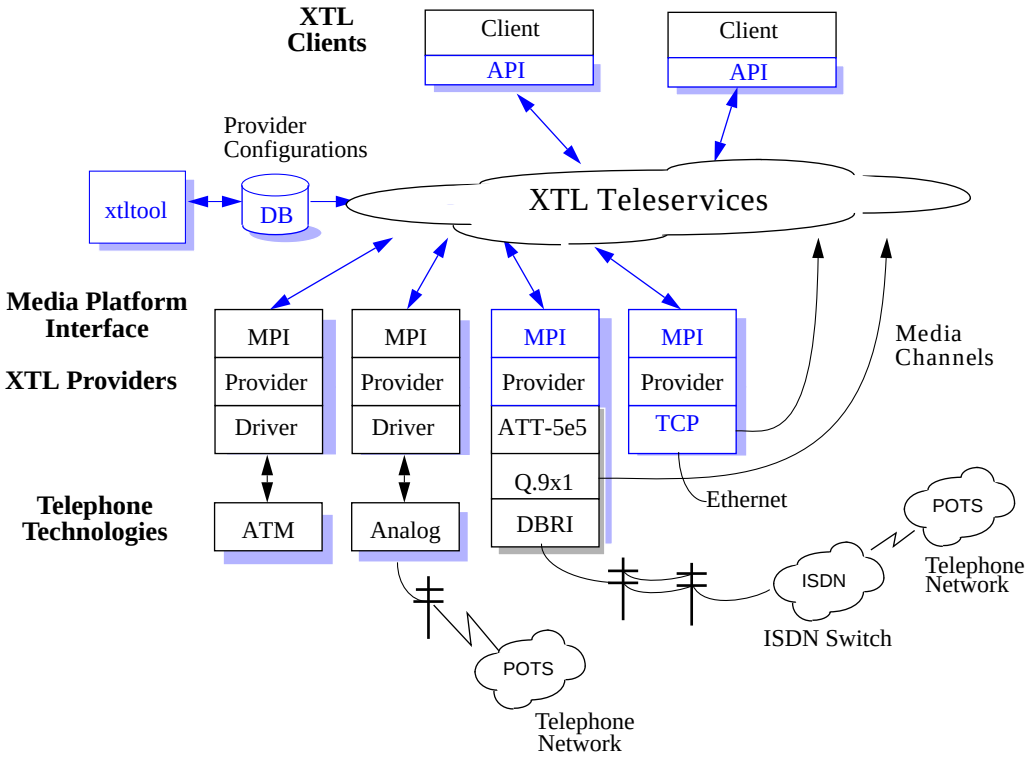


Figure 1-1 XTL Platform Architecture

In the SunXTL framework, the MPI acts as an intermediary between your provider and the rest of the SunXTL platform. Consequently, all communication between SunXTL clients and your provider is relayed through the MPI. However, you can simplify your view of the provider framework during your programming to the one shown in Figure 1-2, which shows only two entities of concern: an abstract entity called the *client* and the underlying technology (telephone device).

The MPI (implemented as the XTLP library) is the interface to the client. All communication between the client and the provider occurs through messages, which conveys call request and indications. The programming model dictates that clients make requests for services to your provider through the MPI, and your provider indicates whether or not it can satisfy such requests by sending indications back to the client through the MPI. When a client makes a request, the MPI invokes a corresponding callback method on an object; when the request has been serviced, the object invokes another method to indicate the success or failure of fulfilling the request. See “Requests and Event Indications” on page 9 for additional description of requests and indications.

The client entity encapsulates all the parts of the SunXTL platform above the MPI. It is the client that makes requests for the services of your provider and receives the indications your provider sends. The client abstraction is important because it allows you to think of requests as originating from a single source, even though several clients may be making requests.

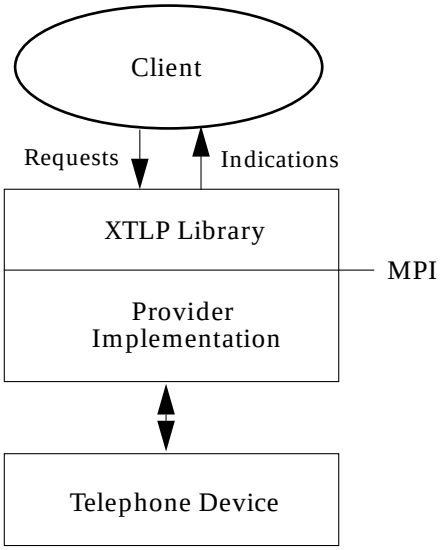


Figure 1-2 Provider Programming Framework

1

The XtlP *Classes*

There are four main classes derived from the XtlP base class. Figure 1-3 shows the XtlP class hierarchy.

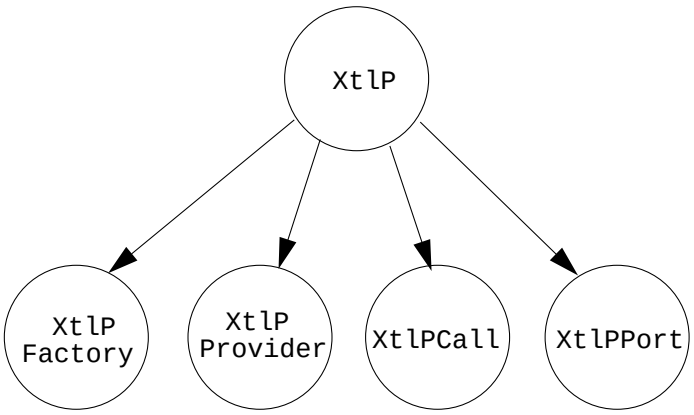


Figure 1-3 XtlP Class Hierarchy

Note – The classes in the XTLP library are not multithread safe.

The XtlP class and its subclasses are described in Chapter 5, “XtlP Classes.” The XtlPFactory, XtlPProvider, and XtlPCall classes contain virtual methods called *request methods* that you must implement. A request method is a callback handler whose implementation satisfies some provider service that a client may request. Callback methods that connect, put on hold, or disconnect a phone call are examples of request methods. The derived XtlP classes contain callback methods that you must override with code that implements the functionality of your provider.

There are also classes and data types that you can use directly without deriving them. Table 1-1 shows the other classes and data types you will use in the process of writing a provider.

Table 1-1 Additional MPI Classes and Data Types

Classes	Description
Utility Classes	
`XtlByteArray`	A reference-counted byte array. See *page 53*.
`XtlString`	A reference-counted string type. See *page 55*
`XtlKVList`	An ordered list structure used to pass parameters between objects. See *page 57*.
Dispatcher Classes	

`dpSLDispatcher`
`dpXtDispatcher`
`dpXVDispatcher`
	Event dispatchers (derived from the InterViews™ library) for use in various user interface environments. See *page 64*.

`dpIOHandler`	An abstract I/O handler for managing events on a file descriptor (this class is also taken from the InterViews library); however, the `dpIOHandler` must be derived. See *page 68*.
Global Data Types	
`XtlFormat`	Data type to describe the format of data being transmitted over a call. See *page 70*.
`XtlCallReference`	A unique identifier assigned to a call. See *page 75*.

1

Provider Framework and Concepts 2

The MPI is one layer in the larger SunXTL architecture. Each layer in the architecture is modular in that the specifics of each layer are invisible to the other. This allows you to program at the provider level without in-depth knowledge of the other layers. For a complete discussion of the architecture, see the *Sun XTL 1.1 Architecture Guide*.

This chapter describes the overall provider framework, helps you to understand the relationships between the `XtlP` classes, and provides several typical call scenarios that can serve as useful examples for writing providers.

The Provider Framework

The provider exists between the MPI and the telephone device, as shown in Figure 1-1 on page 2. Above the MPI is a client entity that makes requests on your provider for *call management* services. Conceptually, the client represents everything above the provider. The provider is only aware of the client above and the device below. The basic function of the provider is to accept *requests* from the client and to send *indications* that convey the results of requests back to the client. In other words, the provider is a consumer of requests and a producer of indications.

So far we have used the term *provider* to name the enabling piece of software that allows a client (encapsulated in the client entity) to manage phone calls using a particular underlying technology. However, providers involve several subtly different and overlapping parts. To better understand the provider framework, it helps to be familiar with the terminology used in this guide:

- Provider
  A high-level term used to describe the software entity that manages a telephone resource; a resource is the device or underlying technology that you wish SunXTL clients to use to communicate.
- Provider object
  A C++ object that is instantiated from a class derived from the `XtlPProvider` class. A provider object has request methods (to carry out requests) and indication methods (to send events to the client). To define the behavior of a provider object, its request methods must be overridden with provider-specific code.
- Provider process
  The address space in which one or more provider objects exist.
- Provider family
  A collection of providers that run in the same provider process and share a common *provider family name*; you define the provider family name in the *template file* in the provider package. Provider families are useful when a provider process needs to support multiple physical devices and where it would be convenient to have individual providers manage each device.
- Provider package
  A collection of files, primarily a provider executable and a provider template file, that is installed by the end user to make use of your specific device or underlying technology.

In general, providers manage calls, which are represented by `XtlPCall` objects. A provider can control one or more calls, which only exist in the context of the provider; a call is always associated with a provider.

The Provider Programming Model

SunXTL providers are event driven in a manner similar to X Windows System clients. In such a programming model, objects wait for *stimuli* from external sources and perform an action to carry out a request. The SunXTL dispatcher library provides the necessary classes to support this event-driven programming model; see “Using the Event Dispatcher” on page 64.

For an SunXTL provider, an external stimulus may be a client request or a state change in the underlying telephone technology as shown in Figure 2-1. The provider sends information back to clients (that is, it informs clients of changes in the state of the client, provider, or call objects) by sending the client one of the following indications:

- Event indications through `event_ind()`
- Error indications through `error_ind()`
- Provider-specific extension indications through `extension_ind()`

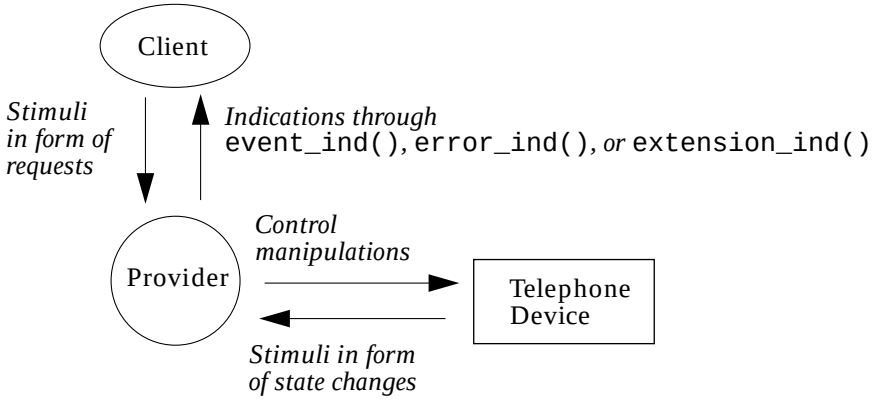


Figure 2-1 Sources of Stimuli to a Provider

Requests and Event Indications

`XtlP` classes communicate with the client through requests and indications as described in “Role of Providers in SunXTL Teleservices” on page 1. Requests from the client cause the MPI to invoke the callback methods of an `XtlP` object. After the provider services a request, the object notifies the client of its success or failure through its indication methods. Your provider program must be able to accept requests from the client and send indications to notify the client of various events. To determine how your provider handles requests and sends indications, you must override the methods in the `XtlP` classes. These classes are described in Chapter 5, “XtlP Classes.”

Relationships Between `XtlP` *Classes*

The primary relationships between `XtlP` classes can be seen in Figure 2-2. Within a process, there is an `XtlPFactory` object that instantiates provider objects; in turn, the provider objects instantiate call objects, which represent individual teleservices connections.

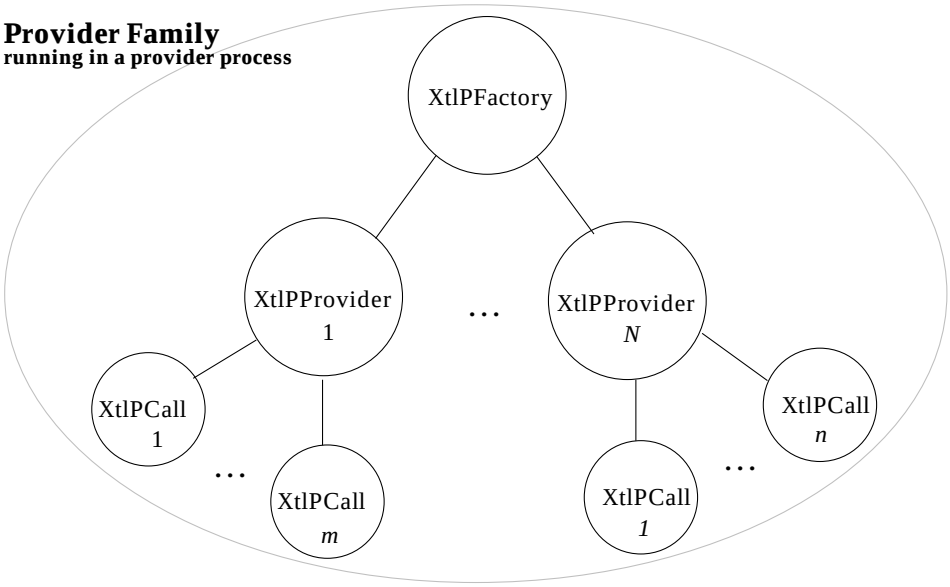


Figure 2-2 Provider Object Instantiation Relationships

The more subtle relationships and behavior require further description and a better understanding of other events that can come into play. The following sections describe those relationships.

XtlP *Classes—Factories, Providers, and Calls*

The main classes in the MPI library are: `XtlPFactory`, `XtlPProvider`, `XtlPCall`, and `XtlPPort`. In writing your provider, you must create subclasses from each of the base classes and implement the callback methods for each class. In general, each class has the following types of methods:

- State slots methods (set and get methods)
- Request callback methods
- Indication methods

Setting and Getting State Slots

State slots hold the attributes of an `XtlP` object and in a call object, they describe the complete state of a call. Each class has a collection of `set_`*slotname*`()` and `get_`*slotname*`()` methods that allow you to set and retrieve the state slots of an object; the *slotname* in the method name corresponds to the name of the state slot. However, certain slots, such as the `call_status` slot, are read-only and therefore do not have a corresponding set method.

In invoking these methods, you do not need to be aware of the implementation of the slots; if you like, you can view slots as member variables of the class. However, the type of the slot value is defined in the signature (argument list) of the slot methods. Specific state slot methods for each class are described in “State Slots” on page 81.”

Receiving Requests From the Client

In the provider programming model, `XtlP` objects receive requests from the client and return indications to the client. Requests are serviced by the callback methods in `XtlP`-derived objects. When the client makes a request, the MPI invokes the appropriate request callback method in your objects. For example, when the client requests a new call object, the MPI invokes the `create_call_req()` callback as implemented in your `XtlPProvider` object.

All request callbacks are pure virtual methods in the base `XtlP` classes. Therefore, you must supply provider-specific implementations for each request callback when you derive from the `XtlPFactory`, `XtlPProvider`, and `XtlPCall` classes.

Sending Indications to the Client

When call states change, errors occur, or extension messages need to be passed to the client, providers must notify the client through one of the following indications:

- Event indications

  Event indications are sent using the `event_ind()` method. Changes in the provider changes a call's state slots. To inform the client of the change, the appropriate state slot must be set and `event_ind()` called; the client will not see the state slot change until your provider calls `event_ind()`. Therefore, your provider is required to call `event_ind()` whenever it changes the state slot of a call, provider, or factory; `event_ind()` is a method relative to a call, provider, or factory object.

- Error indications

  Error indications are sent using the `error_ind()` method when the provider detects an error—such as errors in allocation, protocol, or parameter specification.

  If your provider cannot satisfy a request because of such errors, it should call `error_ind()` on the appropriate object. Additionally, an impatient client may make duplicate requests on a provider. If a request is already satisfied by the current state of the provider, `error_ind()` should be called with the error parameter set to `ERROR_REQUEST_CURRENTLY_SATISFIED`.

- Extension indications

  Extension indications are sent using the `extension_ind()` method to pass an `XtlKVList` containing provider-specific extension values.

Call Status and Transitions

When making or receiving a call, a call object travels a path of call states. These states are defined by a call object's call status. The status of a call is important to the client because the client acts based on the current status of a call object.

Note – The use of the terms *state* and *status* in this chapter may be confusing at first. In telephony vernacular, calls have a state; that is, a call can be idle, on hold, conferenced, and so on. However, in the programming abstraction of a call, there are several attributes that describe the state of a call object—status

being one of several (the others are described in “Setting and Getting State Slots” on page 11). For that reason, the term status is used in place of the more general term, state.

To change the status of a call, the client sends a request to the provider. The provider in turn manipulates the telephone device in an attempt to satisfy the request. When the request is completed, the provider sends an indication to the client. If the request was successfully serviced, the status of the `XtlPCall` object is changed, and not before.

Figure 2-3 shows some of the possible status transitions of a call. Starting from an unknown status, an incoming call takes the left path while an outgoing call takes the right path. As the provider moves from one status to the next, the appropriate indication event is generated, such as `CREATE_EVENT` or `INCOMING_EVENT`. Table 2-1 on page 15 defines the possible call status values.

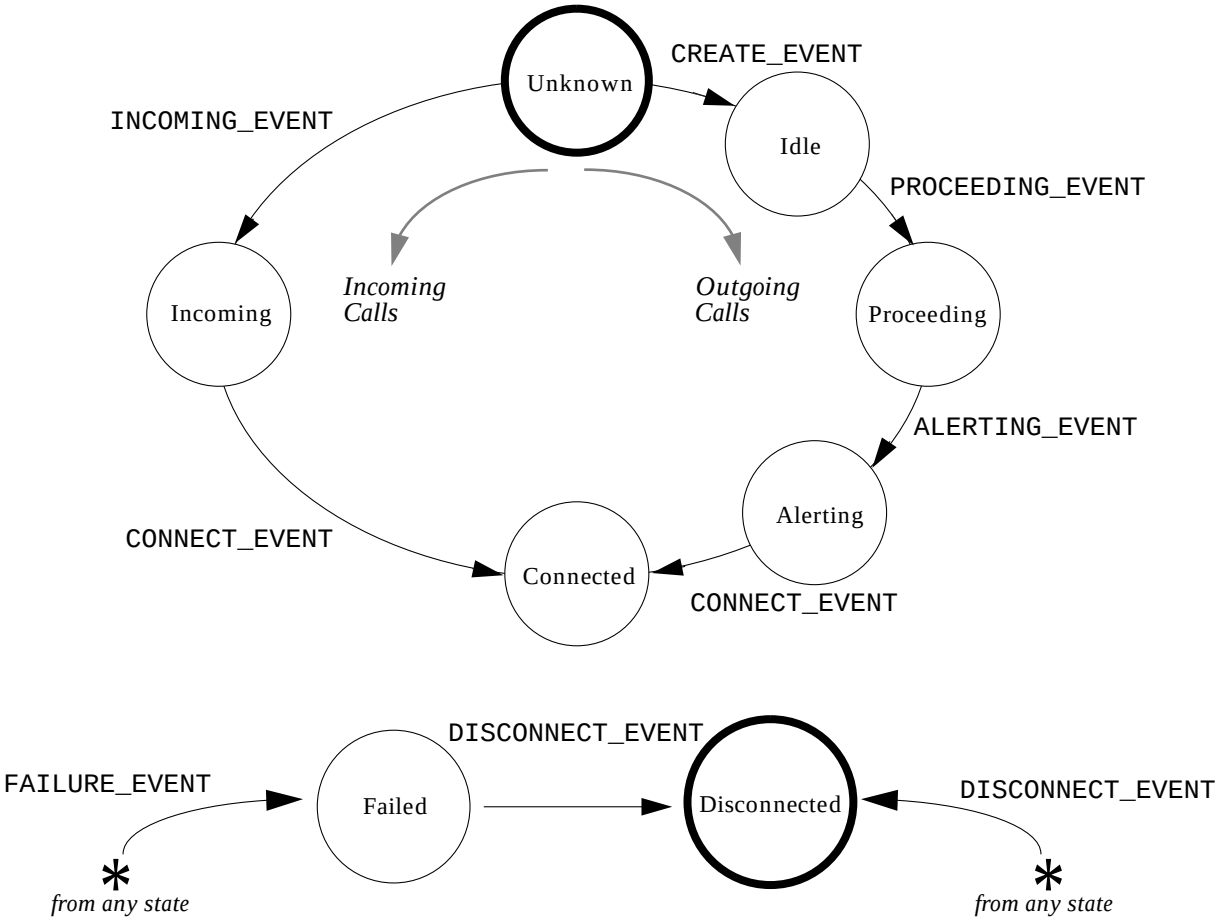


Figure 2-3 Normal Call Status Transitions

Table 2-1 Call Status Enumerations

Call Status	Meaning
UNKNOWN	When an XtlPCall object is created, it starts in an unknown status. At this point the call object is neither incoming nor outgoing.
INCOMING	A remote party is requesting a connection (incoming call).
IDLE	For outgoing calls, after a call object is created, a CREATE_EVENT event is sent to the client and the call is ready to receive a connection request.
PROCEEDING	A complete address has been passed to the call object and an attempt to connect to the remote party is in progress.
ALERTING	The remote party has been notified of the connection attempt; that is, the remote phone is ringing.
CONNECTED	An end-to-end connection has been established; that is, the remote party has answered.
FAILED	A connection attempt has failed. There may be several causes, such as a busy condition, a switch error, or an incomplete address argument. For that reason, this state can be entered from any of the other states. The call then enters the DISCONNECTED state where it is destroyed.
DISCONNECTED	The connection has been terminated and the call object is ready to be destroyed. You may not reuse this call object to make another call.
CONFERENCED	Another call has been conferenced into the current call.

In the normal course of making or receiving a call, a call object can be expected to follow a normal course of status transitions. Of course, other events may occur that cause the transition to deviate; for example, if the call is busy or unexpectedly disconnected. To handle these situations and to ensure that abnormal transitions do not occur (such as going from an INCOMING status to an ALERTING status), the MPI validates each transition so that in a given status, only certain requests may be sent to the provider and only certain event indications may be sent to the client.

Table 2-2 shows the valid client requests that a provider can expect to receive for each call status. Table 2-3 presents a corresponding matrix that shows the valid indication events that can be sent to the client for each call status, and the resulting status transition, if any. The MPI layer enforces these transitions and generates an `EXCEPTION_PROTOCOL_VIOLATION` exception if an inappropriate transition is attempted; exceptions are described in “Exception Codes” on page 78.

Table 2-2 Valid Requests for Call Status

	Call Status[1]								
Requests	Unkn	Idle	Proc	Alrt	Fail	Conn	In	Dis	Conf
CONNECT_REQ	-	Y	-	-	-	-	-	-	-
ADD_TO_ADDRES_REQ	-	Y	-	-	-	-	-	-	-
ANSWER_REQ	-	-	-	-	-	-	Y	-	-
ALERT_REQ	-	-	-	-	-	-	Y	-	-
DISCONNECT_REQ	-	Y	Y	Y	Y	Y	Y	-	Y
REDIRECT_REQ	-	-	Y	Y	Y	Y	Y	-	Y
TRANSFER_REQ[2]	-	-	Y	Y	Y	Y	Y	-	Y
CONFERENCE_REQ[2]	-	-	Y	Y	Y	Y	-	-	Y
DROP_REQ	-	-	-	-	-	-	-	-	Y
HOLD_REQ	-	Y	Y	Y	Y	Y	Y	Y	Y
UNHOLD_REQ	-	Y	Y	Y	Y	Y	Y	Y	Y
CONFIGURATION_REQ	-	Y	Y	Y	Y	Y	Y	Y	Y
EXTENSION_REQ	-	Y	Y	Y	Y	Y	Y	Y	Y

NOTES:

(1) The call status enumeration equivalents are: Unkn=UNKNOWN, Idle=IDLE, Proc=PROCEEDING, Alrt=ALERTING, Fail=FAILED, Conn=CONNECTED, In=INCOMING, Dis=DISCONNECTED, Conf=CONFERENCED.

(2) The call associated with this request has its status in {Proc, Alrt, Conn, In, Fail, Conf} for TRANSFER_REQ or in {Proc, Alrt, Conn, Fail, Conf} for CONFERENCE_REQ. Furthermore, the call argument is likely to be in an active state (that is, its media_channel_available slot is set to B_TRUE); however, the API does not guarantee this. If the call is inactive, your provider should still attempt to transfer the call, otherwise it should return from the method.

(3) A "-" entry in this table means that the MPI guarantees that the request will never be sent in the given state.

Table 2-3 Valid Indication Events for Call Status Transitions

Events	**Current Call Status**								
	Unkn	Idle	Proc	Alert	Fail	Conn	In	Dis	Conf
`CREATE_EVENT`	Idle	-	-	-	-	-	-	-	-
`INCOMING_EVENT`	In	-	-	-	-	-	-	-	-
`PROCEEDING_EVENT`	-	Proc	-	-	-	-	-	-	-
`ALERTING_EVENT`	-	-	Alert	-	-	-	-	-	-
`CONNECT_EVENT`	-	Conn	Conn	Conn	-	-	Conn	-	-
`FAILURE_EVENT`	-	Fail	Fail	Fail	-	Fail	Fail	-	Fail
`DISCONNECT_EVENT`	-	Dis	Dis	Dis	Dis	Dis	Dis	-	Dis
`INFO_EVENT`	-	Idle	Proc	Alert	Fail	Conn	In	Dis	Conf
`TRANSFER_EVENT`	-	-	Proc	Alert	Fail	Conn	In	-	Conf
`CONFERENCE_EVENT`	-	-	Conf	Conf	Conf	Conf	-	-	Conf
`REDIRECT_EVENT`	-	-	Proc	Alert	Fail	Conn	In	-	Conf
`DROP_EVENT`	-	-	-	-	-	-	-	-	Conf

NOTE: This table shows the resulting status after an event. A "-" entry means that a `EXCEPTION_PROTOCOL_VIOLATION` exception will be sent to the client and the call will stay in the same state.

Interfacing to the Underlying Telephone Technology

The MPI library uses an event dispatcher to dispatch control to I/O handlers, which manage file descriptors; see "Using the Event Dispatcher" on page 64, which describes the event dispatcher and I/O handlers in detail.

Your provider should use the dispatcher to interface to the underlying telephone technology. The provider framework assumes that your provider communicates with the underlying telephone technology through a file descriptor, which is used to set up, control, and tear down telephone calls. If your provider does not, you must create such an interface. You can do so by creating a `dpIOHandler` subclass and using the dispatcher's `link()` method to associate an instance of your subclass with a file descriptor. Your

implementation of the `dpIOHandler` then allows your provider to communicate with the underlying technology. An example class derived from a `dpIOHandler` is presented in "Starting Your Provider Process."

To create an I/O handler, create a subclass from the `dpIOHandler` base class and define read, write, and/or exception callback handlers for a file descriptor. The dispatcher's `link()` method allows you to associate your `dpIOHandler` with a specific file descriptor. The dispatcher calls the appropriate callback handler for a given file descriptor when any of these conditions arise:

- Data becomes available on the file descriptor
- A write will not block on a file descriptor
- There is an exception on the file descriptor

When your provider receives stimulus from either the client or the underlying technology, it should spend as little time as possible in the callback and return to the dispatcher loop. For example, when your provider receives a `connect_req()`, it should initiate the outgoing call and immediately return to the dispatcher. It should not block and wait for call progress information from the underlying technology. Rather, as stated above, your provider should create a `dpIOHandler` and file descriptor to handle stimuli from the underlying technology and link the `dpIOHandler` and file descriptor to the dispatcher. When the underlying technology stimulates the provider (that is, presents call progress information), the `dpIOHandler`'s `inputReady()` callback handler is called.

Starting Your Provider Process

Given this event-driven programming model and the dispatcher library routines, the main routine of a typical provider should look similar to:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <xtl/xtlp/xtlp.h>

// The sim_factory.h header file declares a class SimFactory that
// you have derived from XtlPFactory. (There are also
// sim_provider.h and sim_call.h header files which declare
// SimProvider, derived from XtlPProvider and SimCall

#include "sim_factory.h"
```

```
void main(int argc, char *argv[]) {

// Declare err to hold exception values from MPI library calls

XtlP::Exception err;

// Get the provider family name from the environment to be
// passed to the SimFactory constructor.

XtlString family_name(getenv("XTL_FAMILY_NAME"));

// Create the SolarisLive dispatcher.
 dpDispatcher::instance(new dpSLDispatcher);
 dpDispatcher& d = dpSLDispatcher::instance();

 // Verify the Xtl Database

 if (xtl_db_verify() < 0) {
    printf(stderr, "%s: could not verify xtl database\n", argv[0]);
    exit(1);
 }
 // Create a factory object
SimFactory factory(err, family_name);

if (err != XtlP::EXCEPTION_SUCCESS) {
    fprintf(stderr, "Could not create XtlPFactory
object...exiting\n");
    exit(1);
 }

while(B_TRUE)
    d.dispatch();
}
```

Creating Providers

When a provider process is started, it instantiates a factory object and blocks in a dispatch loop until it receives stimuli from either the client or the underlying technology. Typically, the factory object receives a request from the client to create a new provider. The process of creating a provider follows these steps:

1. **Implement the** `create_provider_req()` **callback in the** `XtlPFactory` **object. The MPI will invoke this method when your provider is executed.**

2. **Implement the** `create_provider_req()` **callback. This method constructs a new provider object and performs any provider-specific initialization.**

3. **After the provider object is created, the** `create_provider_req()` **method sends a** `CREATE_EVENT` **indication by calling** `XtlPFactory::event_ind()`**.**

Your `create_provider_req()` callback should look similar to:

```
void SimFactory::create_provider_req(const XtlString& name,
XtlKVList&)
{
    XtlP::Exception exception;
    XtlKVList null_args;
    SimProvider *provider = new SimProvider(exception, *this, name);

    if ((provider == NULL) ||
        (exception != XtlP::EXCEPTION_SUCCESS)) {
        printf("SimFactory::create_provider_req ERROR is %d\n",
               exception);
        delete provider; // It's ok to delete a NULL pointer
        error_ind(exception, ERROR_UNKNOWN, CREATE_PROVIDER_REQ,
                 name, null_args);
        return;
    }

    if(provider != NULL)
        provider->event_ind(exception, XtlPProvider::CREATE_EVENT,
                        XtlP::CAUSE_NORMAL, null_args);
    }
```

The `create_provider_req()` method instantiates a provider object. Because the class for the instantiated provider is derived from the `XtlPProvider` class, the constructor makes the MPI aware of the existence of that object, and allows the MPI to invoke the object's methods to service requests from the client.

Once the provider object is successfully constructed, the provider process must send an `event_ind(CREATED_EVENT)` to inform the client of the event; otherwise, the provider object will not receive any requests from the client.

A typical provider constructor looks similar to:

```
SimProvider::SimProvider(
    Exception& err,
    XtlPFactory& factory,
    const XtlString& name):XtlPProvider(err, factory, name)
{
    // Provider-specific Initialization code ...
}
```

Note – As you might expect, communicating with an underlying technology is highly dependent on that underlying technology. Consequently, as the provider writer, you might instantiate and register a `dpIOHandler` with the dispatcher on a per-provider-basis, rather than having a global `dpIOHandler` for all providers, as done in this example. Whether you do so or not depends on the interface to the underlying technology.

Performing Call Operations

Once a provider is instantiated, the client may use the provider to control calls. This section describes various scenarios for initiating outgoing calls, receiving incoming calls, disconnecting calls, and accessing services (hold, transfer, conference, and drop available to a call). Additionally, a scenario for manipulating the data stream is given.

Note – The figures in this section illustrate the flow of requests and indications that occur during each operation. However, be aware that the method parameters (such as `event_ind()`) shown are not complete, but highlight the critical information that is passed between the MPI and the provider.

Making an Outgoing Call

Figure 2-4 illustrates the exchange of messages and events that occur in the process of making an outgoing call. The following sections provide code examples of how to handle each stage of the process.

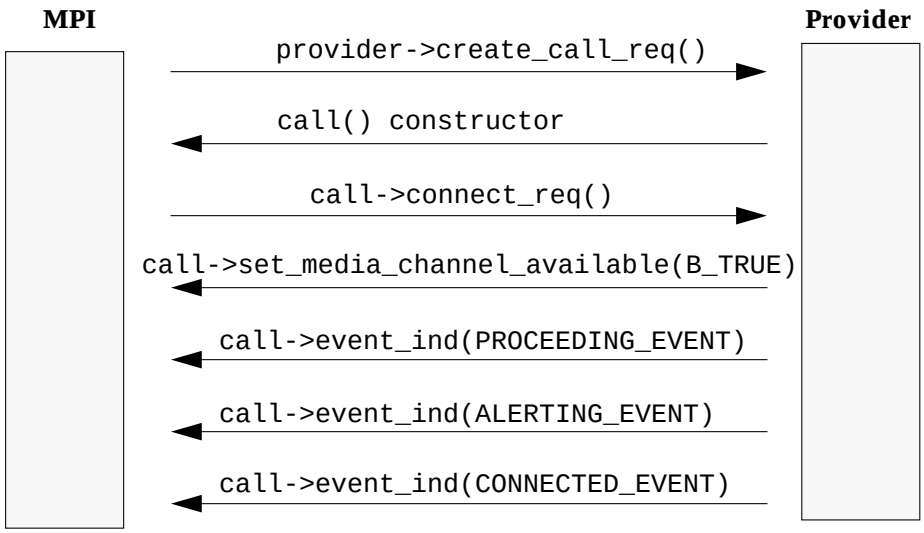


Figure 2-4 Outgoing Call Scenario

Creating a Call

When a client requests an outgoing call, the client and/or MPI determines which provider to use for the call. The MPI then calls the selected provider's `create_call_req()` callback method. As the provider writer, you must supply the implementation of the callback in your derived provider class. A typical `create_call_req()` looks similar to:

```
void SimProvider::create_call_req(XtlKVList& args)
{
    XtlPCall::Exception err;
    SimCall* call = new SimCall(err, *this);
    if ( (call != NULL) && (err == XtlPCall::EXCEPTION_SUCCESS)) {
        call->event_ind(err,
            XtlPCall::CREATE_EVENT,
            CAUSE_NORMAL,
            XtlNullKVListC);
    } else {
        delete call;
```

```
        XtlPProvider::Exception perr;
        error_ind(perr, ERROR_RESOURCE_NOT_AVAILABLE,
                  CREATE_CALL_REQ,
                  XtlNullKVListC);
        }
}
```

You should test exception parameter for `event_ind()` and `error_ind()` for `XtlP::EXCEPTION_SUCCESS`; if some other exception is returned, you should handle the exception condition appropriately.

The `create_call_req()` routine constructs a call object for the call. Because the class for the instantiated call is derived from the `XtlPCall` class, the instantiation of the call object implicitly *registers* the existence of the call with the MPI. Once an outgoing call is constructed, the provider must send an `event_ind(CREATE_EVENT)` to inform the client of the successful creation of an outgoing call. A typical constructor for your derived call class looks similar to:

```
void SimCall::SimCall(XtlP::Exception exception,
                      const XtlPProvider& xtlp_provider){

        // supply provider-specific code to initialize and
        // prepare the provider for the call
}
```

Starting an Outgoing Call

Once a call object is constructed, it receives a `connect_req()` from the MPI when the client wants to initiate an outgoing call. The details of an implementation for `connect_req()` are highly dependent on the underlying telephone technology and so the provider-specific code is deliberately absent in this example:

```
void SimCall::connect_req(
    const XtlAddress& local,
    const XtlAddress& remote,
    const XtlFormat& format_name,
    XtlKVList& args)
{
// check the state of the call by calling get_xxx()
// send messages down to the device to start up your call.
}
```

Monitoring Call Progress

Presumably, after the provider initiates an outgoing call, the provider receives call progress information from the underlying technology. Typical call progress information includes indications of when:

- Enough addressing information has been received to complete a call (call proceeding)
- The remote party associated with a phone call is alerting (ringing)
- The call has been answered by the remote party (connected)

The provider informs the MPI and client of the call progress through the call object's `event_ind()` method.

The underlying technology provides call progress information in the form of callback routines registered by the provider. From these callbacks, the provider informs the MPI (and client) of call progress through the `event_ind()` method.

For example, if a call is proceeding, the upstream callback invokes the call object's `event_ind()` method as follows:

```
event_ind(exception,
        XtlPCall::PROCEEDING_EVENT,
        CAUSE_NORMAL,
        XtlNullKVListC);
```

where `exception` is an output parameter of type `XtlP::Exception`, which indicates any problems in calling `event_ind()`; `XtlPCall::PROCEEDING_EVENT` is a literal in the `XtlPCall::Event` enumeration; `CAUSE_NORMAL` indicates a normal event; and `XtlNullKVListC` is an empty `XtlKVList`.

If there are changes in the state of the call, in addition to the call progress status, the appropriate state slots in the call object should be set before calling `event_ind()`. For example, if the media channel for the call is available when the provider is informed that the call is proceeding, the method `set_media_channel_available()` should be called. For example:

```
example_call->set_media_channel_available(exception, B_TRUE);
```

Note that if the provider detects that the state of the call changes but there is no appropriate event for `event_ind()`, the event may be set to `XtlPCall::INFO_EVENT`.

Receiving Incoming Calls

Figure 2-5 shows a typical exchange of messages and events when a provider receives an incoming call.

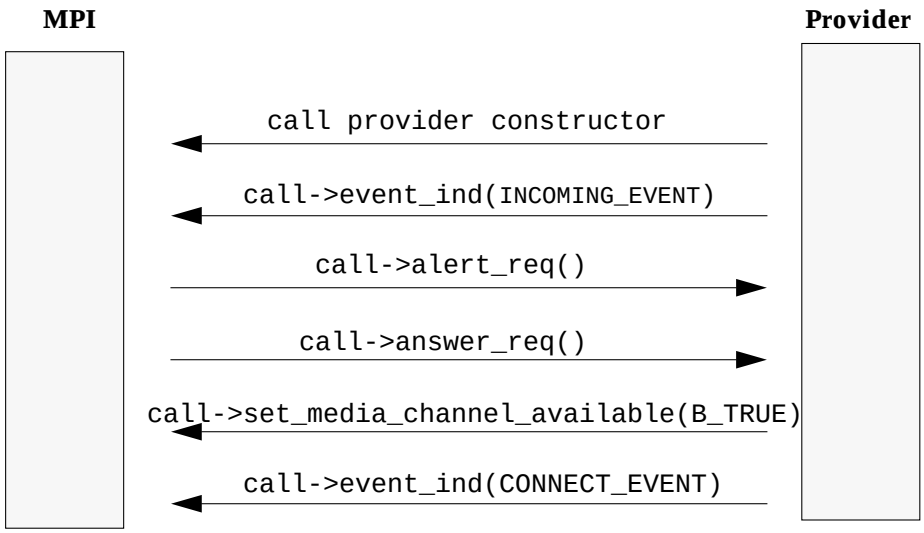


Figure 2-5 Incoming Call Scenario

When the underlying telephone technology detects an incoming call, it informs the provider process through the callback handler associated with the underlying technology. To inform the client of the incoming call, you need to construct a call object for the call and send an `event_ind(INCOMING_EVENT)`. Hence, somewhere within your callback routine, you need code similar to:

```
my_incoming_handler() {
XtlPCall::Exception err;
SimProvider*   sim_provider;

// get the provider object associated with the incoming call
sim_provider = get_provider();

SimCall* call = new SimCall(err, *sim_provider);
if ( (call != NULL) && (err == XtlPCall::EXCEPTION_SUCCESS)) {
```

```
     call->event_ind(err,
     XtlPCall::INCOMING_EVENT,
     CAUSE_NORMAL,
     XtlNullKVListC);
 } else {
    // add code to execute error handling routine
    //and delete call object
    return;
}
```

Once the MPI is informed of the incoming call, the provider should return to the `dispatch()` loop and wait for further stimulus to indicate how the call should be handled. Typically, the MPI informs the provider that a client has been informed of the incoming call by calling the `alert_req()` callback for the call. Your callback for the alert request should look similar to:

```
void SimCall::alert_req(XtlKVList& args) {
    // Add code to inform the underlying technology that this
    // "phone is ringing"
}
```

After the callback executes, the provider returns to the `dispatch()` loop. If the client wants to answer the call, the MPI invokes the call object's `answer_req()` callback. The callback for the request looks similar to:

```
void SimCall::answer_req(XtlKVList& args)
{
    XtlP::Exception err;

    // Insert code to inform the underlying technology that
    // this phone wants to answer the call

    // Send up an event if this was successful
    event_ind(err, CONNECT_EVENT, CAUSE_NORMAL, XtlNullKVListC);

    // Check to see if the event_ind() call was successful
     if(err != XtlPCall::EXCEPTION_SUCCESS) {
        // execute error handling routine
        return;
    }
}
```

After `answer_req()` executes, the provider returns to the `dispatch()` loop. If all goes well, the underlying technology informs the provider that it has connected with the calling party. As usual, this is handled through the input

IOHandler callback associated with the underlying technology. Within the call object, you should write code to inform clients of the availability of the data channel and of the end-to-end connection. The code looks similar to:

```
// From within the call object:
set_media_channel_available(exception, B_TRUE);
event_ind(exception, XtlPCall::CONNECT_EVENT,
          XtlP::CAUSE_NORMAL, XtlNullKVListC);
```

Note that in this example, the provider has been able to determine that the media channel associated with the call is available, and it informs the MPI and clients of this fact by setting the call's `media_channel_available` slot.

Disconnecting Calls

Calls can be disconnected for a variety of reasons. A disconnect can be requested by the underlying technology (usually when the remote party "hangs up"), by the client, by the MPI, or by the provider itself. Figure 2-6 shows a typical call disconnection scenario.

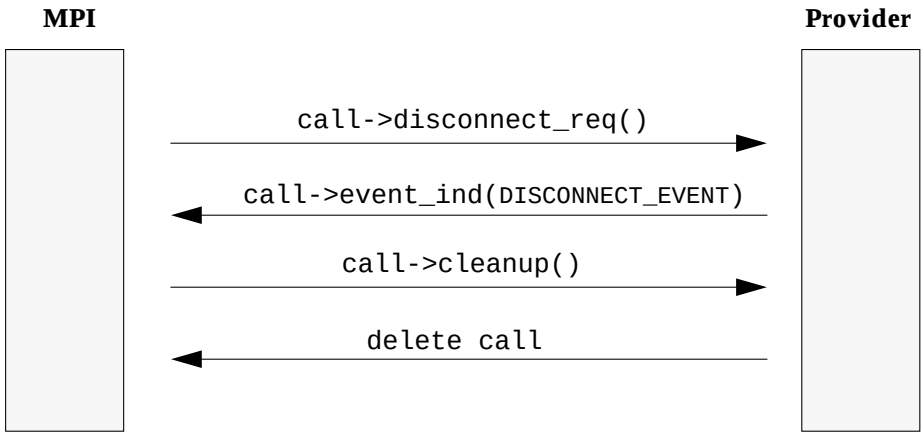


Figure 2-6 Call Disconnection Scenario

When the client requests to disconnect a call, the MPI invokes the call's `disconnect_req()` callback. Your implementation of this callback should look similar to:

```
void SimCall::disconnect_req(XtlKVList& args) {
    // add provider-specific code to initiate the release of the call
}
```

When a provider detects that a call has been successfully disconnected (usually though the callback associated with the underlying technology), the provider should inform the MPI of this by sending an `event_ind()`:

```
sim_call->event_ind(exception,
                    XtlPCall::DISCONNECT_EVENT,
                    XtlP::CAUSE_NORMAL,
                    XtlNullKVListC);
```

After the disconnect event is sent to the MPI, the MPI eventually invokes the call's `cleanup()` callback. Once the call's cleanup method has been called, the MPI will no longer call any of the call's callbacks. At this point, the provider usually deletes the call object associated with the call:

```
SimCall::cleanup() {
    if (call_has_been_cleared) {
            delete this;
    } else {
            // initiate clear process
            delete this;
    }
}
```

Note – The MPI can call the `cleanup()` callback on a call at any time. In particular, it is not necessary that the MPI call the `disconnect_req()` callback or wait for the `event_ind(DISCONNECT_EVENT)` before calling `cleanup()`. Because the MPI specifies that no call request callbacks will be called after `cleanup()` is called, the provider must now take complete control of the call. In particular, once the `cleanup()` callback is called, the provider is responsible for clearing the call.

Furthermore, the provider may delete a call at anytime. However, it is recommended that the provider clear the call, send an `event_ind(DISCONNECT_EVENT)` and wait for a `cleanup()` callback before deleting a call.

Putting Calls On and Taking Calls Off Hold

Figure 2-7 shows how calls are put on hold and taken off hold. A `hold_req()` is a request to release the media channel associated with a call while `unhold_req()` reacquires the media channel for the call's use.

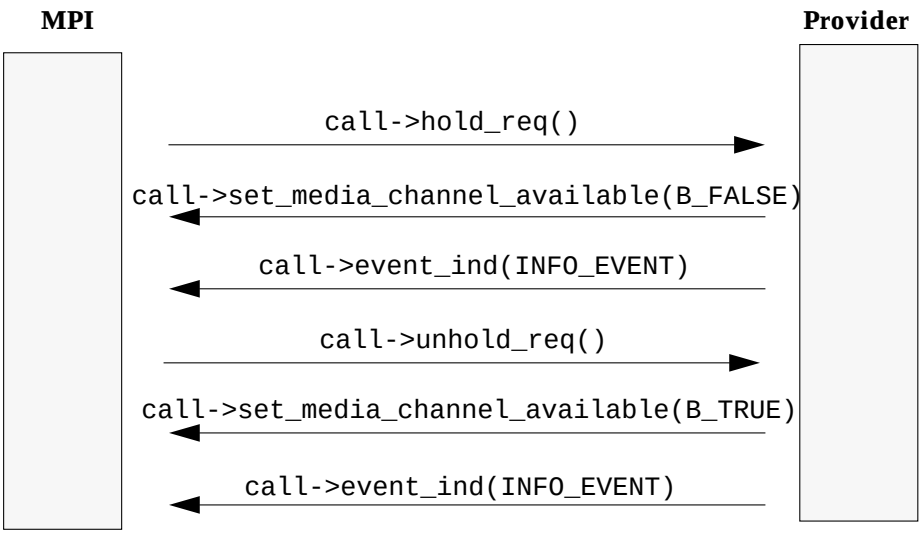


Figure 2-7 Call Hold and Unhold Scenario

Your `hold_req()` callback should look similar to:

```
void SimCall::hold_req(XtlKVList& args) {
    // add provider-specific code to request call be put on hold
}
```

When the provider determines that the call is on hold (the media stream channel for the call has been released), the provider should invoke the call object's methods:

```
mycall::set_media_channel_available(exception, B_FALSE);
mycall::event_ind(exception,
           INFO_EVENT,
           HOLD_REQ,
           XtlPCause::CAUSE_NORMAL,
           XtlNullKVListC);
```

If a client wants to take a call off hold (that is, reacquire a media channel for the call), the MPI invokes the `unhold_req()` callback:

```
void SimCall::unhold_req(XtlKVList& args) {
    // add provider-specific code to request call be taken off hold.
}
```

When a media channel is again available for the call, your provider should:

```
set_media_channel_available(exception, B_TRUE);
event_ind( exception,
           INFO_EVENT,
           XtlPCause::CAUSE_NORMAL,
           XtlNullKVListC);
```

Transferring a Call

A request to transfer a call is a request to establish a communication path between the remote party associated with the call and the remote party associated with the transferee parameter (an `XtlCallReference &`) of the transfer request. The transfer makes it appear as if a call was made from one remote party to the other. Typically, once a call is successfully transferred, both this call and the call associated with the transferee are disconnected. The code for your `transfer_req()` callback should be similar to:

```
void SimCall::transfer_req(
    const XtlCallReference& transferee,
    XtlKVList& args)
{

    XtlP::Exception err;
    XtlPCall *transferee_call=0;

    // _provider points to the call's provider object
    _provider->get_call_object(err, transferee, transferee_call);

    if (err != XtlP::EXCEPTION_SUCCESS) {
        error_ind(err,
                ERROR_SERVICE_NOT_AVAILABLE,
                TRANSFER_REQ,
                XtlNullKVListC);
        return;
    }

    // Add code here to transfer the call and
```

```
    // Notify the client that the calls have been transferred and
    // that both call objects are now in the disconnected state.

    transferee_call->event_ind(err,
                          DISCONNECT_EVENT, CAUSE_NORMAL,
                          XtlNullKVListC);
    event_ind(err, TRANSFER_EVENT, CAUSE_NORMAL, XtlNullKVListC);
    event_ind(err, DISCONNECT_EVENT, CAUSE_NORMAL, XtlNullKVListC);
    delete transferee_call;
    delete this;
}
```

If the transferee is a call that belongs to the same provider as this call, the call to `_provider->get_call_object()` gets the call. Because `get_call_object()` returns a pointer to an `XtlPCall`, the transferee is *narrowed* to an `ExampleCall`. Note that as of the last indication sent to the transferee object, the media channel for it was available. Furthermore, the `transfer_req()` will be received by the provider only if the `call_status` is either `PROCEEDING`, `ALERTING`, `CONNECTED`, `FAILED`, or `CONFERENCED`.

Conferencing a Call

A request to conference a call is a request to establish a communication path between the remote party associated with the call and the remote party associated with the conferee parameter of the conference request (an `XtlCallReference`). Figure 2-8 shows the call conference scenario.

Conferencing a call is different from transferring a call in that the communication path between the call and both remote parties remain intact, whereas transferring a call destroys the communication path between the call and both parties. Consequently, the media stream associated with the call is a shared communication medium between the local party (client), the call's remote party, and the other remote party that is to be conferenced.

In conferencing, communication occurs over a *handle* that is agreed upon between the switch and the underlying technology; this handle is a mechanism of the switch and is not controlled by the SunXTL platform. For example, in ISDN, the call reference value represents the handle for a call.

In the context of this section, the object whose `conference_req()` method was invoked is referred to as the call object that represents the call being conferenced. The call object associated with the conferee parameter is called the *conferee object*. The call associated with the conferee object (and the conferee parameter) is referred to as the *conferee*.

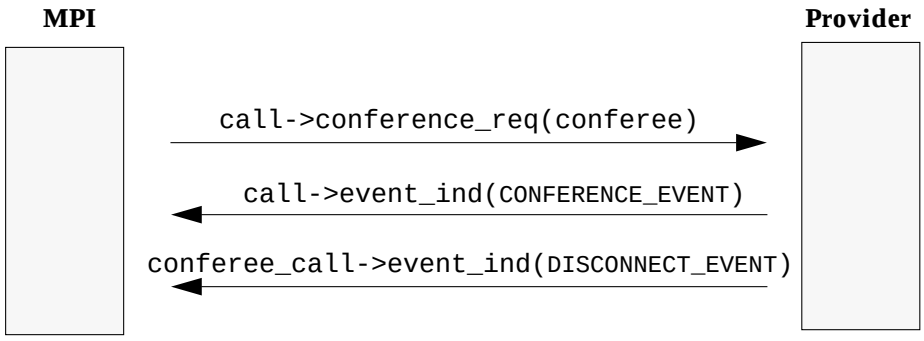


Figure 2-8 Call Conference Scenario

Guaranteeing Conference Indications

Providers have the responsibility of guaranteeing that conferences appear on the same call object through which the conference request was made. The `CONFERENCE_EVENT` is always indicated on the call object that received the conference request. If the underlying technology you are using cannot guarantee that, then your provider must make it appear as if it does. The `CONFERENCE_EVENT` should never be sent up on the deleted conferee call object.

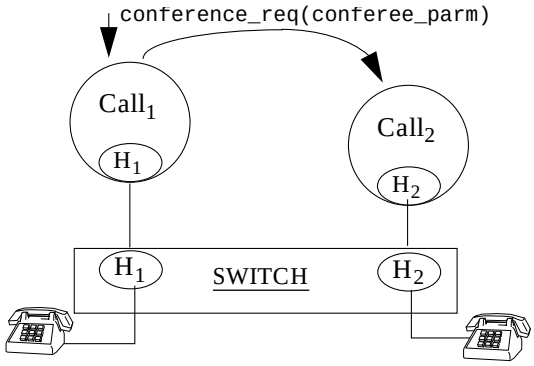
Assume that once a call is successfully conferenced, the underlying telephone technology arbitrarily clears either the call or the conferee. If it is the call that is cleared, then all attributes of the call object and the conferee object must be swapped (with the exception of the media stream configuration and the `call_status`, but including all states in the derived class in which the call objects were instantiated). However, the conferee's media channel is not disconnected.

When a call object is successfully conferenced, you invoke `event_ind(CONFERENCE_EVENT)` on the call object and invoke `event_ind(DISCONNECT_EVENT)` on the conferee object. Figure 2-9 shows the swapping process.

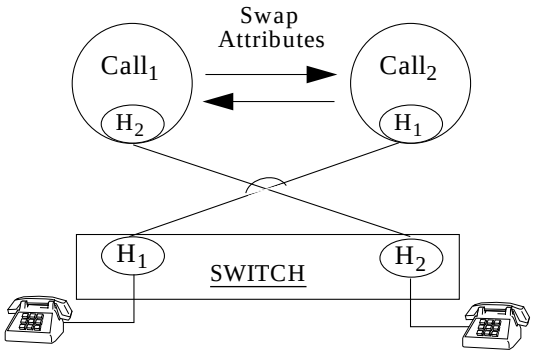
Note that as of the last indication sent to the conferee object before the `conference_req()` was sent to the call object, the media channel for the conferee was available. Further, the `conference_req()` of a call is invoked by the MPI only if the `call_status` for the call is either `PROCEEDING`, `ALERTING`, `CONNECTED`, `FAILED`, or `CONFERENCED`.

Swapping the attributes of the call object and conferee object is required to ease the programming burden on SunXTL client writers. If a client writer wants to create a conference (that is, have a single communication channel to multiple parties), a call is designated as the conference. All other calls are merely added to the conference. Acknowledgement of the conference request is always acknowledged on the call object making the request for the conference. Further, the configuration of the media stream associated with the conference does not change as new calls are added to the conference. Hence, reconfiguration of the data stream is never needed as calls are added to a conference.

1. A client requests a conference call through *Call*$_1$, which causes the MPI to invoke the `conference_req()` callback on the call object *Call*$_1$ with a conferee parameter *Call*$_2$. The calls initially occur over the switch's handles H_1 and H_2, respectively.



2. The switch decides to use H_2 for the conference, and discard H_1. Because the request came through *Call*$_1$, the provider must maintain *Call*$_1$ by swapping attributes and call handles.



3. Once the calls are conferenced, the call handle H_1 is destroyed, *Call*$_2$ sends up a disconnect event, and *Call*$_1$ sends up a conference event. The calls are now conferenced through the *Call*$_1$ call object.

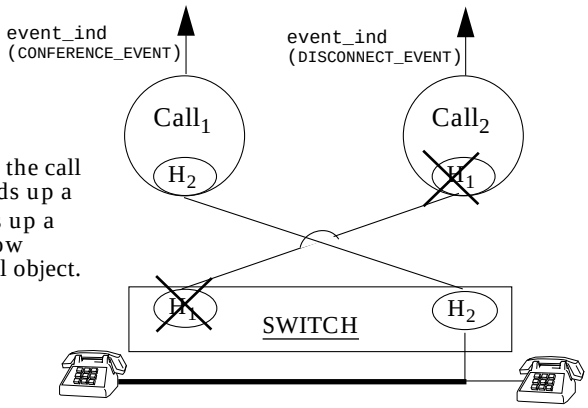


Figure 2-9 Conference Call Scenario With Handle Swapping

The code for your `conference_req()` callback should be similar to:

```
void SimCall::conference_req(
                  const XtlCallReference& conferee,
                  XtlKVList& args) {

    XtlP::Exception exception;
    SimCall *  other_call;
    XtlPCall * temp_call;

    // _provider is a pointer to the provider object, saved from
    // the arguments passed to the SimCall constructor
    _provider.get_call_object(exception, transferee, temp_call);

    if (exception != XtlP::EXCEPTION_SUCCESS) {
        this->error_ind(exception,
                        XtlP::ERROR_INVALID_ARGUMENT,
                        XtlPCall::CONFERENCE_REQ,
                        XtlNullKVListC);
        return;
    }

    other_call = (SimCall*) temp_call;// narrow to SimCall

    // add code to conference the call
}
```

If the conferee object belongs to the same provider as the call object, the procedure call to the `_provider.get_call_object()` method will get the call. Because `get_call_object()` returns a pointer to an `XtlPCall`, the conferee object is narrowed to an `ExampleCall`.

If the conference request succeeds, your code should implement the following pseudo code:

```
SimCall * call = call_that_survives;
SimCall * other_call = call_that_will_be_cleared;

if (other_call == call_that_requested_the_conference) {
        SimCall temp_call;

        // swap attributes of call and other call
        // exception data stream configuration and call_status
        temp_call.attributes = call->attributes;
        call->attributes = other_call->attributes;
        other_call->attributes = temp_call.attributes;
```

```
}

call->event_ind(CONFERENCE_EVENT);
other_call->event_ind(DISCONNECT_EVENT);
delete other_call;
```

Dropping a Call From a Conference

A conference can be viewed as a call in which several parties communicate over a common medium that has been set up by your provider. Input by any participant in the conference is available to all participants in the conference. Many underlying telephone technologies allow parties to be dropped from the conference—typically the last party added. Consequently, the `drop_req()` callback allows clients to drop parties from a conference.

The MPI does not provide explicit means for a client to specify which party should be dropped from the conference. If your technology supports such capabilities, the `XtlKVList` on the `drop_req()` should be used to allow clients to make such a specification. If your technology only supports the dropping of the last party, there is some ambiguity as to which party will be dropped because of the potential swapping of call object attributes when the conference was set up; refer to “Guaranteeing Conference Indications” on page 33. Under these circumstances, you may not have control over which party is dropped.

Note – If a remote party chooses to drop from a conference, the switch will disconnect the party, however, your call object might not be notified of this event. This is switch-specific behavior.

Figure 2-10 shows the drop call scenario. Your `drop_req()` callback should implement the following pseudo-code:

```
void SimCall::drop_req(XtlKVList& args) {

    // supply code to drop the last party in a conference

}
```

If the drop succeeds, your provider then calls `event_ind()` on the conference call:

```
event_ind(exception,
        DROP_EVENT,
        XtlPCause::CAUSE_NORMAL,
        XtlNullKVListC);
```

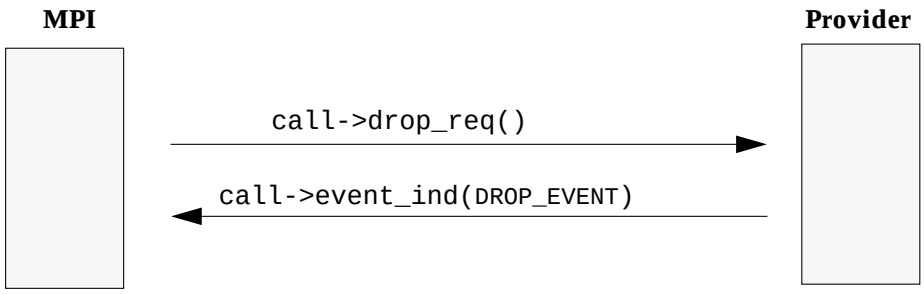


Figure 2-10 Drop Call Scenario

Configuring Media Channels

If a client wants to configure the media channel associated with a call, the MPI invokes the call's `configuration_req()` callback. If the media channel for the call is available, your provider should try to put the media channel into the requested configuration. If the media channel for the call is not currently available, your provider must check to see if the requested configuration differs from the current configuration. If so, the provider should cache the request, set the call's configuration slot, and send an `event_ind(CONFIGURATION_EVENT)` to indicate how it will attempt to configure the media channel when it becomes available.

When the media channel becomes available, the provider should attempt to set up the media channel as prescribed by the configuration slot. If the provider cannot configure the media channel as prescribed by the configuration slot, the provider should set the configuration slot to the current configuration and send an `event_ind(CONFIGURATION_EVENT)`.

Hence, your callback for the `configuration_req()` should look similar to:

```
void SimCall::configuration_req(XtlKVList& args) {
    XtlKVList configuration;
    boolean_t media_channel_available;
    XtlP::Exception exception;

    get_media_channel_available(exception,
                                media_channel_available);
    configuration = validate_configuration(exception, args);

    if (exception != XtlP::EXCEPTION_SUCCESS) {
        error_ind(err,
                XtlP::ERROR_INVALID_ARGUMENT,
                CONFIGURATION_REQ,
                XtlNullKVListC);
    }
}
```

Note that the exception output parameter for `event_ind()` and `error_ind()` should be checked, and, if not equal to `XtlP::SUCCESS`, the exception should be handled appropriately.

Presenting Media Channels to the Client

In designing your provider, you may want to allow the client access to the media channel associated with a call so that it can read, write, redirect, filter, or perform some manipulation on a call's data. A provider can give the client control of the media channel by using the following procedure:

1. Construct an `XtlPPort` object.
2. Use the call object's `update_port_ind()` method to register the port.
3. Configure the call's media stream using the call object's `set_configuration()`method.
4. Send an event by calling the call's `event_ind()` method.

Your provider must be able to access a call's data as a STREAMS® stream. When your provider wants to give clients access to the data, it must construct an `XtlPPort` object. The argument to the constructor is the file descriptor of the stream associated with the call's data. Constructing the `XtlPPort` object gives the MPI access to the data associated with the stream through the standard STREAMS mechanisms. Your provider can revoke the MPI's access to the data by destroying the `XtlPPort` object.

Between the time your provider constructs an `XtlPPort` object and the time it destroys it, it is not valid for your provider to manipulate (for example, read or write) the stream associated with the `XtlPPort`. Furthermore, if multiple `XtlPPorts` are constructed for a given stream head, it is invalid for your provider to manipulate the associated stream until the destructor for each port object has been called.

Note – `XtlPPorts` do not distinguish between stream heads that are opened with different modes. Hence, if your provider wants to provide read and write access to a call's data, it must do so by creating a single `XtlPPort` for the stream, and then the stream can be opened in read/write mode.

The code you write should look similar to:

```
int example_fd;
XtlPPort * example_xtlp_port;
XtlP::Exception exception;
example_fd = open("/dev/streams_data_channel", O_RDWR);

example_xtlp_port = new XtlPPort;
    .   .   .
// until delete, access to fd is not valid
    .   .   .
delete example_xtlp_port;
```

Constructing an `XtlPPort` for a stream head only informs the MPI of a STREAM that it can manipulate. The STREAM must now be associated with a call. This is done through the `update_port_ind()` method of a call. Hence, once the `XtlPPort` for the call's data is constructed you must call:

```
SimCall * sim_call
sim_call->update_port_ind(exception, example_xtlp_port);
```

In addition to constructing the `XtlPPort` and calling `update_port_ind()` for the call, your provider must update the data stream configuration for the call. To do so:

1. **Create an** `XtlKVList` **for the configuration with the appropriate** `<XtlConfigInputK, XtlConfigStreamC>` **and/or** `<XtlConfigOutputK, XtlConfigStreamC>` **pair(s).**
2. **Configure the media stream using** `set_configuration()`**.**
3. **Send an** `event_ind(INFO_EVENT)`**.**

For example, you might write code that looks similar to:

```
XtlKVList configuration;
sim_call->get_configuration(exception, configuration);
configuration->add(XtlConfigInputK, XtlConfigStreamC);
configuration->add(XtlConfigOutputK, XtlConfigStreamC);
sim_call->set_configuration(exception, configuration);

sim_call->event_ind(exception,
                    XtlPCall::INFO_EVENT,
                    XtlPPCall::CAUSE_NORMAL,
                    XtlNullKVListC);
```

The ordering of activities described above is the way your provider will usually sequence activities to present data stream access to the MPI. However, this need not be the case. For example, if a client asks to configure a media channel that is currently unavailable (see “Handling Media Channel Configurations” on page 97), your provider might choose to call `set_configuration()` for the desired media channel and send an `event_ind(INFO_EVENT)` immediately, even though the media stream is not available. This allows the client to proceed without waiting on this event. When the media stream does become available, it merely constructs the port and calls `update_port_ind()`. Setting the configuration and calling `event_ind()` to inform the MPI of the media stream configuration is not necessary because this was previously done in the preconfiguration dialog.

2

Creating a Provider Package 3

This chapter describes how to create a provider executable and how to package your provider product for end-users.

Note – To use the SunXTL libraries, your development system must be current. For SPARC systems, use C++ 3.0.1 in SPARCompilers 2.0.1. For x86 systems, use C++ 3.0.1 in ProWorks 2.0.1.

Writing a Provider Program

The process of writing a provider is straightforward and well defined. It involves creating subclasses from the `XtlPFactory`, `XtlPProvider`, and `XtlPCall` classes, and implementing their pure virtual functions. You can begin this writing process from nothing or from a set of files that contains generic provider code.

Quick Start to Writing a Provider

The most time-consuming part of writing a provider is implementing code for the pure virtual functions in your subclasses. For that reason, files containing generic skeleton code to create any provider is supplied to help you begin the writing process quickly. The generic code declares the necessary data types and

`main()` function—you need only fill in the empty methods with functionality to define your provider's behavior. Table 3-1 shows the files you can use to begin writing a provider.

Table 3-1 Generic Provider Code Files

File	Description
`call.h`	Defines a subclass named `Call`.
`call.cc`	Declares empty methods for `Call`.
`factory.h`	Defines a subclass named `Factory`.
`factory.cc`	Declares empty methods for `Factory`.
`provider.h`	Defines a subclass named `Provider`.
`provider.cc`	Decleares empty methods for `Provider`.
`debug.h`	Includes `<assert.h>` and `<syslog.h>` headers and defines `SYSLOG()`
`main.cc`	Defines a signal handler, initializes the dispatcher, and instantiates a factory object to start off the provider.
`Makefile`	A makefile to create a provider executable. This makefile can be customized by changing the variables: `PROG_NAME`, `PROJECT_CCFLAGS`, `PROJECT_CPPFLAGS`, `PROJECT_LDFLAGS`, and `PROJECT_LDLIBS`.

Writing a Provider from the Beginning

If you want to create your provider program from the beginning, follow these steps:

1. **Include the header file** `<xtl/xtlp/xtlp.h.>`**.**
   This header file contains all necessary MPI header files in one:

   - `xtlp_globals.h`
   - `xtlp_call.h`
   - `xtlp_factory.h`
   - `xtlp_provider.h`
   - xtlp_port.h
   - `bytearray.h`
   - `kvlist.h`
   - `xtldb.h`

2. **Include the necessary** `bytearray.h`**,** `kvlist.h`**, and** `xtldb.h` **header files.**
3. **Include** `<Dispatch/iohandler.h>` **and the appropriate dispatcher header file** `<Dispatch/sldispatcher.h>`**,** `<Dispatch/xtdispatcher.h>`**, or** `<Dispatch/xvdispatcher.h>`**.**
4. **Derive the** `XtlPCall` **class; see “XtlPCall Methods” on page 88**
   a. **Override the** `cleanup()` **method.**
   b. **Override the request callback methods.**
5. **Derive the** `XtlPProvider` **class; see “XtlPProvider Methods” on page 85.**
   a. **Override the** `cleanup()` **method with code to delete any private structures, reset the hardware state, and so on.**
   b. **Override the request callback methods. You must supply your implementations for the** `create_call_req()` **and** `extension_req()` **callbacks.**
6. **Derive the** `XtlPFactory` **class; see “XtlPFactory Methods” on page 83.**
   a. **Override the** `cleanup()` **method.**
   b. **Override the** `create_provider_req()` **method.**
7. **Create the** `main()` **loop.**
8. **Within the** `main()` **loop, instantiate a dispatcher object and define it as the global dispatcher by calling** `dpDispatcher::instance()`**.**
9. **Within the** `main()` **loop, instantiate your** `XtlPFactory` **object; providers and call objects will be instantiated dynamically when requested by the client.**
10. **Enter the event loop for your dispatcher.**

3

Linking to the SunXTL Libraries

Table 3-2 shows the various libraries you should link for each of the classes that your provider uses.

Table 3-2 Library Names and Functions

Name	Function	LD_FLAGS
`libxtlutil.so`	Utility classes (`XtlString`, `XtlByteArray`, `XtlKVList`, and so on)	`-lxtlutil`
`libxtlp.so`	MPI classes library	`-lxtlp`
`libdispatch.so`	Dispatcher and I/O handler classes	`-ldispatch`

Compiling the Provider Simulator

The SunXTL package provides example code in the directory `/opt/SUNWxtl/src/simulator`.

The directory contains a makefile that creates two executables: the provider `xtlp_sun_sim` and a test program `xtl_provider_test`.

Creating a Provider Package

As a provider writer, you must package your provider so that users can install it using `pkgadd(1M)` and configure it using the SunXTL administration tool `xtltool(1)`. You can assume that users have already installed the SunXTL runtime package before they attempt to install your provider package. However, you may not assume that the SunXTL package has been installed in its default path; thus your provider package must be relocatable; see “Creating Relocatable Packages” on page 47. A basic provider package should contain:

- All necessary scripts to install a package (for example, preinstall, postinstall, and prototype description files as used by `pkgadd`)
- A provider executable
- A template file
- Any other provider-specific files and directories

Creating Relocatable Packages

All provider packages should be constructed so that they are fully relocatable in case the SunXTL platform is not installed in its default location. The default location for the SunXTL runtime package is `/opt/SUNWxtl`. However, because system administrators have the freedom to install the package elsewhere, your provider package must be relocatable. For your provider to adapt to different installations, you need to supply a start-up script that can examine its environment and pass that information to your provider executable when it starts.

The start-up script is pointed to by the `XTL_START` key in the provider configuration file and in the template file; Table 3-3 describes the `XTL_START` key. `XTL_START` should always point to an intermediate script that performs any needed setup before actually invoking the provider executable. This allows such environment variables as `PATH` and `LD_LIBRARY_PATH` to be set before the provider is invoked.

For example, the following Bourne shell code uses `pkginfo(1)` to query the location of the SunXTL package. Once the location is discovered, the location of SunXTL libraries is set so that the provider executable can run.

```
XTLB_ROOT=‘pkginfo -r SUNWxtlb 2> /dev/null‘
[ -z "$XTLB_ROOT" ] && XTLB_ROOT=/
[ "$XTLB_ROOT" = / ] && XTLDIR=/opt/SUNWxtl ||
XTLDIR=${XTLB_ROOT}/SUNWxtl

exec env LD_LIBRARY_PATH=$XTLDIR/lib $XTLDIR/bin/xtlp_my_provider
```

Creating Provider Templates

The SunXTL platform uses a template file from which it creates configuration files for providers. The template file contains defaults for several mandatory keys and allows you to define your own provider-specific keys. These keys define provider attributes, such as the provider family name, a version number, paths to documentation, and so on. Table 3-3 shows the keys and values that you must specify in your template file.

To create a template file, use any ASCII editor. The template file is simply an ASCII file containing key-value pairs. To help users configure your provider, you need to supply default values for the keys described in Table 3-3. When

users configure a provider, `xtltool(1)` creates a default configuration file for the provider based on your template. Using `xtltool`, the user can then change the configuration with the aid of documentation that you provide.

Table 3-3 Key-Value Pairs for Provider Template Files

Key Name	Description of Value
`XTL_FAMILY_NAME`	The `XTL_FAMILY_NAME` value is a unique, system-wide name of the provider's provider family.
`XTL_PROVIDER_NAME`	The `XTL_PROVIDER_NAME` value is a unique, system-wide name for the provider. The `XTL_PROVIDER_NAME` value can be changed by changing the provider's primary alias using `xtltool`. See the *Sun XTL 1.1 Administrator's Guide* for information about aliases.
`XTL_START`	The value of `XTL_START` is a fully qualified path name to a start-up script that executes your provider executable, with any necessary parameters; the `XTL_START` value cannot be empty. The script may be a blocking or non-blocking script. At any given time, only one instance of the provider script will be executing. Consequently, the `XTL_START` value must be the same for all providers of the same family. If the SunXTL platform cannot communicate with your provider process, it invokes the `XTL_START` script again. When the SunXTL platform starts your provider, it does so with the following environment: `XTL_FAMILY_NAME=` <family name from configuration file> It is important to note that the `XTL_FAMILY_NAME` value is all that is passed in the environment. This means that any additional environment variables such as `PATH` and `LD_LIBRARY_PATH` must be set by the `XTL_START` start-up script.

Table 3-3 Key-Value Pairs for Provider Template Files

Key Name	Description of Value
`XTL_DOCUMENTATION`	This key is intended for presenting documentation about the provider to the user. The value is a fully qualified path to an executable along with parameters for the executable; `xtltool` provides an `Execute Document Command` button. When the button is pressed, the value of the `XTL_DOCUMENTATION` key is executed. The documentation command is executed with the same environment as `xtltool`.
`XTL_SETUP`	This key is intended for doing arbitrary provider-specific initialization. The `XTL_SETUP` value is a fully qualified path to an executable along with parameters for the executable; `xtltool` provides an `Excute Setup Command` button. When the button is pressed, the value of the `XTL_SETUP` key is executed. The `XTL_SETUP` command is executed with the same environment from which `xtltool` was run.
`XTL_VERSION`	The value of `XTL_VERSION` should reflect the current SunXTL platform release level for which the provider runs; for example, 1.0.
Other provider-specific key-value pairs (optional)	A template file may also contain other default key-value pairs that describe configuration information that is specific to your provider.

Creating Template Files at Installation

If your provider package has dependencies on other software packages, you could write a script that creates a template file to reflect the configuration of the user's host; for example, your script could create a template based on certain environment variables or the location (paths) of other packages. This can be done using preinstallation scripts that are used in the `pkgadd` installation process.

Template File Format

Your documentation should fully describe how users configure a provider; that is, the documentation should describe all provider-specific key-value pairs that the provider recognizes as well as the required keys.

3

When your provider package is installed, a template file must be copied to, or created in, the `/etc/xtl/templates` directory. The name of the template file should be <*provider_family_name*>`.template`, where <*provider_family_name*> is a unique, system-wide name.

The key-value pairs in the template file are represented in ASCII. The value portion of the key-value pair can only be of type `XtlString`. A line in the template file is either a key-value pair, a comment (a line that begins with a "#"), or white space (a line consisting only of the characters in `[ \t\n]`—space, tab, or newline characters). If a line is a key-value pair, it is essentially a string (a sequence of ASCII characters with no white space) followed by a colon (:), followed by zero or more strings (the value). Formally, a line representing a key-value pair in the template file has the following form:

```
<kv_pair> == space key space ":" space value "\n"
```

where:

```
<space> == [ |\t]*
<key>   == [any non-space ASCII character except for colons]+
<value> == NULL or
        [a non-space ASCII character][an ASCII character except \n]*
```

An example of a line in a template file representing a key-value pair would be:

```
XTL_START : /opt/MYCOxtl/bin/my_provider -a -b -c
```

where the key is `XTL_START` and the value, an `XtlString`, is `/opt/MYCOxtl/bin/my_provider -a -b -c`.

Thus, when the user installs your provider, it will be configured with this key-value pair. Code Example 3-1 shows a template file example for a 5e5 provider.

```
# Key               Value
# ---               -----
XTL_VERSION:        1.0
XTL_START:          /opt/SUNWx5e5p/scripts/xtlp_sun_5e5.start
XTL_CLEANUP:        RELOCDIR/OPT_OR_USR/scripts/xtlp_sun_5e5.cleanup
XTL_FAMILY_NAME:    xtlp_sun_5e5
XTL_PROVIDER_NAME:  xtlp_sun_5e5
XTL_SETUP:          /opt/SUNWconn/bin/isdntool
XTL_DOCUMENTATION:  pageview /opt/SUNWx5e5p/doc/xtlp_sun_5e5_doc.ps


# Provider-specific key-value pairs
DEFAULT_SPEAKER:           SPEAKER
DEFAULT_MICROPHONE:        MICROPHONE
B1:                        /dev/isdn/0/te/b1
B2:                        /dev/isdn/0/te/b2
DBRI_MANAGEMENT_CHANNEL:   /dev/isdn/0/mgt
UMUX:                      /dev/xtl/umux
SWITCH_TYPE:               5E5
TERMINAL_TYPE:             B
NUMBER_OF_CALL_APPEARANCES:3
NAI:                       0
```

Code Example 3-1 Sample Provider Template

3

Utility Classes 4

The utility classes provide the data structures and event-handling objects that are used throughout the API and MPI libraries. These utility classes define the structures necessary to convey simple and aggregate data, which can be passed among Xtl objects; these containers of data include the classes `XtlByteArray`, `XtlString`, and `XtlKVList`. The remaining utility classes are `dpDispatcher` and `dpIOHandler`, which handle file descriptor I/O events; if you have worked with the InterViews library, these classes should be familiar because they are derived from those classes.

Using `XtlByteArray`

An `XtlByteArray` object is a byte array structure with the addition of convenient operators, such as assignment, length count, and equality comparison of array elements. An `XtlByteArray` is typically used to hold `XtlAddress` values and provider-specific values in `XtlKVList` objects. Table 4-1 shows the methods provided by the `XtlByteArray` class.

4

Table 4-1 `XtlByteArray` Methods (from `bytearray.h`)

Method	Description
Constructors	
`XtlByteArray();`	Constructs a zero-length `XtlByteArray`.
`XtlByteArray(const XtlByteArray& a);`	Constructs a copy of a given `XtlByteArray`.
`XtlByteArray(const char* bytes, u_int len);`	Constructs an `XtlByteArray` from a buffer of `len` bytes.
`XtlByteArray(const char* string);`	Constructs an `XtlByteArray` from a null-terminated string.
Operators	
`const char* bytes() const;`	Returns the contents of an `XtlByteArray` as a pointer to a buffer.
`u_int length() const;`	Returns the number of bytes in the array.
`const char* operator()() const;`	Like `bytes()`, this operator returns the contents of the `XtlByteArray` as a string. This operator is provided to be consistent with `XtlString::operator()`.
`boolean_t operator== (const XtlByteArray&) const;`	Compares two `XtlByteArrays` for equality.
`boolean_t operator!= (const XtlByteArray&) const;`	Compares two `XtlByteArrays` for inequality.
`XtlByteArray& operator= (const XtlByteArray&);`	Assigns contents of one `XtlByteArray` to another.

`XtlByteArray` offers some flexible ways of assigning and initializing elements in the array. An `XtlByteArray` can be shared and passed to functions without concern for memory management because it is reference counted. Code Example 4-1 shows several ways to initialize and manipulate an `XtlByteArray`.

Note – There are no data alignment guarantees when using `XtlByteArray` except that byte order and byte boundaries are preserved. If you need to store structured data in an `XtlByteArray`, you must convert the structure to a byte format first by using an `xdr(3N)` conversion routine; this helps to maintain code portability.

Code Example 4-1 `XtlByteArray` Usage Examples

```
#include <xtl/bytearray.h>

// XtlByteArray Examples

 char buffer[256];
 // pretend buffer was initialized with a 23-byte structured value
 XtlByteArray array(buffer,23);

 // print total size of array and second byte in array
 printf("size=%d, array[1] = %d\n", array.length(),
array.bytes()[1]);

 // an alternate syntax would use operator() instead of bytes()
 printf("size=%d, array[1] = %d\n", array.length(),
array.operator()[1]);
```

Using `XtlString`

An `XtlString` encapsulates a reference-counted string in much the same way as `XtlByteArray` does an array. The main difference is that `XtlString`s are null terminated while `XtlByteArray`s may contain embedded null values. You can assign `XtlString`s in the same manner as char * strings and pass `XtlString`s without de-referencing them. Table 4-2 shows the methods provided by the `XtlString` class. Code Example 4-2 shows some examples of `XtlString` usage.

Table 4-2 `XtlString` Methods (from `bytearray.h`)

Method	Description
Constructors	
`XtlString();`	Constructs an empty `XtlString`.
`XtlString(const char* str);`	Constructs an `XtlString` from a regular string.

Table 4-2 `XtlString` Methods (from `bytearray.h`) *(Continued)*

Method	Description
`XtlString(const XtlString& s);`	Constructs a copy of a given `XtlString`.
`XtlString(const XtlByteArray& s);`	Constructs an `XtlString` from an `XtlByteArray`. Only bytes up to the first null are copied.
Operators	
`const char* bytes() const`	Returns the contents of an `XtlString` as a pointer to a buffer.
`u_int length() const;`	Returns length of an `XtlString`.
`const char* operator()() const;`	Returns the contents of an `XtlString` as a string.
`boolean_t operator== (const XtlString&) const;`	Compares two `XtlStrings` for equality.
`boolean_t operator== (const char*) const;`	Compares `XtlString` to a string for equality.
`boolean_t operator!= (const XtlString&) const;`	Compares two `XtlStrings` for inequality.
`boolean_t operator!= (const char*) const;`	Compares an `XtlString` to a string for inequality.
`XtlString& operator= (const XtlString& str);`	Assigns one `XtlString` to another.

Code Example 4-2 `XtlString` Usage Examples

```
#include <xtl/bytearray.h>

extern "C" int printf(const char *, ...);

void print_xtlstring(const XtlString& str) {
 printf("XtlString='%s'\n", str());
}

main() {
```

Code Example 4-2 `XtlString` Usage Examples *(Continued)*

```
// XtlString Examples

XtlString mystr("foobar");   // construct a string on the stack.

 print_xtlstring(mystr);// pass an xtlstring as an arg.

 print_xtlstring("baz");// construct a temporary string and pass it.
                        // the literal is converted to XtlString.

 mystr = "blat";       // change the value of the variable mystr.

 print_xtlstring(mystr);// pass the new value to print_xtlstring()


 XtlString newstring = "foobar2";    // construct a null XtlString
                                     // then assign a string value

 mystr = newstring;          // make mystr the same as newstring
 print_xtlstring(mystr);

 printf("length=%d\n", mystr.length());// print length of XtlString

 XtlString anotherstring(newstring);// duplicate an XtlString
 XtlString nullstring;       // create a null XtlString
}
```

Using `XtlKVList`

`XtlKVList` objects are used mainly as method arguments to pass parameters in the form of lists. An `XtlKVList` object is an ordered list of *key* and *value* pairs as shown in Figure 4-1; it may also contain other `XtlKVList` objects. In a key-value pair, the key is always an `XtlString` while the corresponding value can be of type `u_long`, `XtlString`, `XtlByteArray`, or a nested `XtlKVList`.

Some characteristics of an `XtlKVList` object are:

- An `XtlKVList` is ordered and traversed sequentially. An internal pointer points to the current position in the list.
- `XtlKVList` objects are reference counted to relieve you of memory management chores. The copy and assignment operators perform lazy copies, so that an actual copy operation only occurs if the list is modified.
- You can `add()` and `remove()` key-value pairs, move to the `first()` pair, `next()` pair, or `reset()` the current pointer position to the beginning of the list. You can also `get()` the value or retrieve the `key()` from a key-value pair.
- Key-value pairs may be removed from the list, relative to the *current position*. The current position is specified by the current pointer, which is a reference to a key-value pair on the list; that key-value pair can also be thought of as the *current key-value pair*.
- Key-value pairs are retrieved in the same order they were added. That is, in a first-in, first-out manner.
- Key-value pairs are always added to the end of the list without changing the current position.
- Removing a key-value pair moves the current pointer to the preceding pair. That is, removing the third pair causes the current pointer to point to the second pair. Removing the first pair puts you at the beginning of the list (the same position in which a `reset()` leaves you).

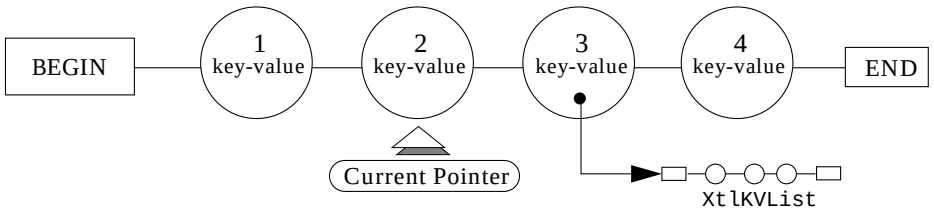


Figure 4-1 `XtlKVList` Structure

The XtlKVList class offers the command methods listed in Table 4-3.

Table 4-3 XtlKVList Methods (from kvlist.h)

Method	Description
`boolean_t add(const XtlString& key, u_long value)`	The add() methods add a specified key-value pair to the end of the XtlKVList structure. The methods differ only in the type of value that is appended. B_TRUE is returned upon success; otherwise B_FALSE is returned.
`boolean_t add(const XtlString& key, const XtlString& value)`	Adds an XtlString value; the XtlString is copied, so the list does not reference the value argument. B_TRUE is returned upon success; otherwise B_FALSE is returned.
`boolean_t add(const XtlString& key, const XtlByteArray& value)`	Adds an XtlByteArray value; the XtlByteArray is copied, so the list does not reference the value argument. B_TRUE is returned upon success; otherwise B_FALSE is returned.
`boolean_t add(const XtlString& key, const char* value)`	Adds a null-terminated string value; the value is stored as an XtlString. B_TRUE is returned upon success; otherwise B_FALSE is returned.
`boolean_t add(const XtlString& key, const XtlKVList& list)`	Adds an XtlKVList to the current list; it does not affect the internal pointers that point to the current key-value pair in each list. B_TRUE is returned upon success; otherwise B_FALSE is returned.
`boolean_t remove()`	Removes the current key-value pair from the list.
`boolean_t get(u_long& val)`	Gets the value of the current key-value pair. If the value is a u_long, val is set to that value and B_TRUE is returned. If the value is of another type, val is undefined and B_FALSE is returned.
`boolean_t get(XtlString& val)`	Gets the value of the current key-value pair. If the value is an XtlString, val is set to that value and B_TRUE is returned. If the value is of another type, val is undefined and B_FALSE is returned.
`boolean_t get(XtlByteArray& val)`	Gets the value of the current key-value pair. If the value is an XtlByteArray, val is set to the value and B_TRUE is returned. If the value is of another type, val is undefined and B_FALSE is returned.

Table 4-3 `XtlKVList` Methods (from `kvlist.h`) *(Continued)*

Method	Description
`boolean_t get(XtlKVList& list)`	Gets the value of the current key-value pair. If the value is an `XtlKVList`, `val` is set to the value and `B_TRUE` is returned. If the value is of another type, `val` is undefined and `B_FALSE` is returned.
`u_long count()`	Returns the number of key-value pairs in the list.
`boolean_t key(XtlString& key)`	Gets the key of the current key-value pair. If there is no current pair (such as at the beginning or end of a list), key is undefined and `B_FALSE` is returned. Otherwise, `key` is a reference to the key and `B_TRUE` is returned.
`boolean_t type(Type& t)`	Returns the type of the value in the current key-value pair. If the current pointer is at the head or tail of the list, `B_FALSE` is returned; otherwise, `B_TRUE` is returned. You can use the type value in `switch()` statements to perform conditional actions. The `Type` enumeration has the values: `ULONG`, `STRING`, `BYTEARRAY`, `KVLIST`.
`boolean_t first()`	This method is a shorthand equivalent to using `reset()` followed by a `next()`. The return code is the return code from `next()`.
`boolean_t first(const XtlString& key)`	This method is a shorthand equivalent to using `reset()` followed by a `next(key)`; the pointer is placed on the first key-value pair whose key matches the `key` argument. The return code is the return code from `next()`.
`void reset()`	Sets the current pointer to the beginning of the list, before the first key-value pair. Note that you need to use `next()` to set the pointer to the first key-value pair. This also means that `get()` and `remove()` will fail after a `reset()` unless a `next()` is first performed.
`boolean_t next()`	Advances the current pointer to the next key-value pair in the list. `B_TRUE` is returned upon success. If you are at the end of the list and there is no next pair, `B_FALSE` is returned and the current pointer moves off the list.

Table 4-3 `XtlKVList` Methods (from `kvlist.h`) *(Continued)*

Method	Description
`boolean_t next(const XtlString& key)`	Advances the current pointer to the next key-value pair that has a `key` equal to the `key` argument. `B_TRUE` is returned upon success. If the list contains no matching pairs after the current pointer, then `B_FALSE` is returned, and the current pointer is positioned at the end of the list.

Table 4-3 `XtlKVList` Methods (from `kvlist.h`) *(Continued)*

Method	Description

`boolean_t subset(const XtlKVList&, XtlKVListCompareFunc = NULL)`	Compares this list with the argument list and returns `B_TRUE` if this list is a subset of the argument. The lists are treated as sets, thus order and duplicates are not considered in the comparison; for example, two lists, `a` and `b`, are set equivalent if `a.subset(b)` and `b.subset(a)` both return `B_TRUE`. In basic use, `subset()` compares the value types `ULONG`, `STRING`, and `BYTEARRAY`; if either list contains embedded lists, the comparison fails.
	
	However, you can specify an optional comparison function to compare lists that have embedded lists. The arguments to your custom comparison function must have the form:
	
	`boolean_t kvlist_compare_fn(`
	`  const XtlString key,`
	`  const XtlKVList& a_list,`
	`  const XtlKVList& b_list)`
	
	where `a_list` is an embedded list in this list and `b_list` is an embedded list from the argument to `subset()`. The comparison function should return `B_TRUE` if `a_list` is a subset of `b_list`.

`void print(int fd)`	Prints the contents of the list in a structured format. The output is directed to the specified file descriptor. See “Creating a Hierarchical XtlKVList” on page 62 for an example of usage and output.
`extern "C" void print_kvlist(` `  const XtlKVList*);`	Like `print()`, this external C function prints the contents of the list in a structured format to standard output (`stdout`). It is intended for use in debuggers that may have difficulty calling `XtlKVList::print()`.

Copying an `XtlKVList`

A program can make a copy of an `XtlKVList` by using either of the following `XtlKVList` methods:

```
XtlKVList(const XtlKVList& r);
XtlKVList& operator=(const XtlKVList& r);
```

Creating a Hierarchical `XtlKVList`

The `add(const XtlString& key,const XtlKVList& list)` method allows you to create `XtlKVList`s that have a hierarchical or recursive structure.

The print routine accommodates hierarchical `XtlKVList`s by displaying the number of elements in the embedded `XtlKVList`, followed by the key-value pairs, which are indented from the previous level. For example, the code:

```
#include <xtl/kvlist.h>

XtlKVList kvlist;
XtlKVList kvlist2;

kvlist.add("key1","val1");
kvlist.add("key2",3);

kvlist2.add("key4","val4");
kvlist2.add("key5","val5");
kvlist2.add("kvlistkey",kvl2);
kvlist2.add("key6",6);

kvlist.add("kvlistkey",kvl2);
kvlist.add("key3","val3");
kvlist.print(1);
```

results in the following output:

```
key="key1" value="val1"
key="key2" value=3
key="kvlistkey" kvlist.count=4
       key="key4" value="val4"
       key="key5" value="val5"
       key="kvlistkey" kvlist.count=2
              key="key4" value="val4"
              key="key5" value="val5"
       key="key6" value=6
key="key3" value="val3"
```

Traversing `XtlKVLists`

A common task your programs need to perform is to traverse and examine the contents of an `XtlKVList`. The following code shows how to traverse a list:

```
XtlKVList kvlist;
Type type;

kvlist.reset();

while (kvlist.next()) {
    kvlist.type(type);

    switch(type) {
        case XtlKVList::ULONG {
            u_long u;
            kvlist.get(u);
            // do something
        }

        case XtlKVList::STRING {
            XtlString string;
            kvlist.get(string);
            // do something
        }

        case XtlKVList::BYTEARRAY {
            XtlByteArray array;
            kvlist.get(array);
            // do something
        }

        case XtlKVList::KVLIST {
            XtlKVList list;
            kvlist.get(list);
            // do something
        }
}
```

Using the Event Dispatcher

The event dispatcher (`dpDispatcher`) and I/O handler (`dpIOHandler`) classes used in the SunXTL libraries are derived from the InterViews library classes. The `dpDispatcher` class works closely with the `dpIOHandler` class,

which is associated with file descriptors. When I/O handlers are linked to a dispatcher (using `dpDispatcher::link()`), the dispatcher polls each I/O handler in round-robin manner. When new data appears on any of the file descriptors, the dispatcher passes control to the I/O handler. In this context, new data means there is new input or output ready on a file descriptor, or that an exception (timer expiration) occurred; see the `select(3C)` man page. Figure 4-2 helps to illustrate these concepts.

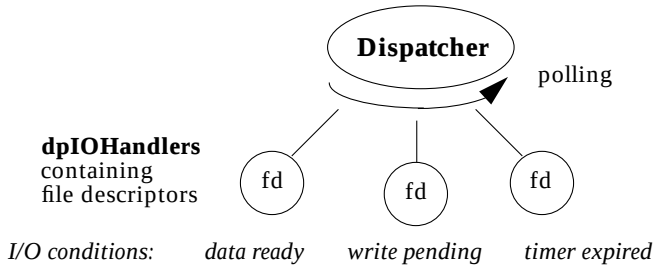


Figure 4-2 `dpDispatcher` and `dpIOHandler` Interactions

Again, the dispatcher used in the SunXTL platform is derived from the InterViews dispatcher class. For SunXTL Teleservices, a subclass of `dpDispatcher` called `dpSLDispatcher` is defined, where `SL` signifies the name Solaris Live! Unlike the standard `dpDispatcher` class, this subclass allows more than 20 file descriptors, and allows you to unlink a `dpIOHandler` from within another `dpIOHandler` without problem.

The dispatcher library is called `libdispatch.so` and should be linked with `-ldispatch` in addition to `-lxtlutil` and `-lxtl` or `-lxtlp`.

4

The standard InterViews dispatcher does not work with SunXTL Teleservices. To program in SunXTL Teleservices, you must use one of the following dispatchers:

- `dpSLDispatcher` for command line environments
- `dpXVDispatcher` for use with XView™
- `dpXtDispatcher` for use with OLIT™ or Motif®)

You should use `dpSLDispatcher` for programs that do not need to interact with the window system (for example, if you are writing providers or command-line-based applications). If you are using XView, use `dpXVDispatcher` from `xvdispatcher.h`. If you are using OLIT or Motif, use `dpXtDispatcher` from `xtdispatcher.h`.

An application only needs one instance of a dispatcher. The static member function, `dpDispatcher::instance()`, is available to create an instance of the dispatcher and then return a reference to it. If a dispatcher already exists, a reference to the existing dispatcher is returned. Table 4-4 shows the `dpDispatcher` class methods.

Table 4-4 `dpDispatcher` Methods (from `dispatcher.h`)

Method	Description

`virtual void link(`
`  int fd,`
`  DispatcherMask mask,`
`  dpIOHandler* ioh)`
	Attaches a `dpIOHandler`, given its file descriptor and a `DispatcherMask`. The `DispatcherMask` describes the I/O conditions that the `dpIOHandler` is interested in, such as whether the file descriptor has new data available for reading. The possible mask values are: `ReadMask`, `WriteMask`, and `ExceptMask`.
	
	When an I/O condition occurs, the `dpIOHandler` is expected to read data from the file descriptor, write data to the file descriptor, or handle the exception depending on the I/O condition.

`virtual dpIOHandler* handler(`
`  int fd,`
`  DispatcherMask mask)`
	Returns the `dpIOHandler` for a given file descriptor.

`virtual void unlink(int fd)`	Detaches the `dpIOHandler` associated with the file descriptor argument.

Table 4-4 `dpDispatcher` Methods (from `dispatcher.h`) *(Continued)*

Method	Description

`virtual void startTimer(`
`  long sec,`
`  long usec,`
`  dpIOHandler* ioh)`
	Starts the timer for the specified `dpIOHandler`. The time specified by `sec` and `usec` is relative; that is, you can tell the timer to expire in five minutes, but you cannot tell it to expire at 5 P.M.

`virtual void stopTimer(dpIOHandler*)`	Stops the timer for the specified `dpIOHandler`.

`virtual unsigned setReady(`
`  int fd,`
`  DispatcherMask)`
	Allows you to artificially set a file descriptor as ready, which triggers a dispatch.

`virtual void dispatch()`	The dispatch routine blocks all registered `dpIOHandler` objects until an event occurs. Internally, `dispatch()` calls the system call `select(3C)`; once `select()` returns, `dispatch()` invokes the appropriate `dpIOHandler` and returns. Your program should loop on `dispatch()` to continuously handle events.

`virtual unsigned dispatch(`
`  long& sec,`
`  long& usec)`
	Calling `dispatch()` with a time value causes `dispatch()` to block until an event occurs on one of its IOHandlers or until the specified time elapses. This is useful if you need to regain control after a fixed period of time.

`static dpDispatcher& instance()`	Returns a reference to the static, global dispatcher object.
`static void instance(dpDispatcher*)`	Installs the specified dispatcher to act as the global dispatcher.

Initializing the Dispatcher

The dispatcher must be initialized before use. The following code shows how to do this:

```
#include <Dispatch/iohandler.h>
#include <Dispatch/sldispatcher.h>

void main() {
    dpSLDispatcher d; // create SolarisLive dispatcher
    dpDispatcher::instance(&d); // install dispatcher instance

    // do other Xtl initialization

    // enter main dispatch loop
```

```
    for (;;) {
        d->dispatch();
    }
}
```

Note – The initialization code for the dispatcher must be called before any other SunXTL code. If it is not, two dispatcher processes are created, but only one will run. As a result, certain internal routines will have registered their handlers with the old dispatcher before the new one is started by your program—this can cause your code to break.

However, for the Xt and XView dispatchers, you should not call `dispatch()` in your dispatch loop. Instead, use the appropriate window toolkit mechanism, such as `xv_main_loop()`.

```
#include <Dispatch/iohandler>
#include <Dispatch/xvdispatcher.h>

void main() {
    // initialize xview
    dpXVDispatcher d; // create specific dispatcher
    dpDispatcher::instance(&d); // install dispatcher instance
    // do other Xtl initialization
    // enter xview notifier loop
    xv_main_loop();
}
```

For a Motif application, do the following:

```
#include <Dispatch/iohandler>
#include <Dispatch/xtdispatcher.h>

void main() {
    XtAppContext context;
    // initialize Xt intrinsics
    // create specific dispatcher
    dpXtDispatcher d(default_app_context);
    dpDispatcher::instance(&d); // install dispatcher instance
    // do other Xtl initialization
    // enter Xt notifier loop
    XtAppMainLoop(context);
}
```

Using dpIOHandler

The dpIOHandler class is used with the dispatcher as described in "Using the Event Dispatcher" on page 64. A dpIOHandler manages read, write, and exception handling operations for a file descriptor. The dispatcher calls a dpIOHandler when the I/O condition for a file descriptor changes.

Note – dpIOHandler objects return values that affect the behavior of the dispatcher. If a dpIOHandler returns a negative value, the dispatcher initiates the unlink() command and ignores the file descriptor. If a positive value is returned, the dispatcher marks the file descriptor as ready and goes through the dispatch loop again. If zero is returned, the dispatcher assumes the callback is finished with the file descriptor, and continues normally.

The dpIOHandler() constructor creates the dpIOHandler object. This object consists of the callback functions: inputReady(), outputReady(), exceptionRaised(), and timerExpired(). The timerExpired() function is called when a timer started with the dispatcher has expired. You should avoid using UNIX timers such as setitimer(2); UNIX timers are asynchronous and can be disruptive if the timer expires during an SunXTL call. Instead, you should use synchronous dpDispatcher timers.

Table 4-5 shows the dpIOHandler class methods. By default the methods are empty, so you need to override the method(s) for which the handler will use.

Table 4-5 dpIOHandler Methods (from iohandler.h)

Callback Method	Description
`virtual int` **`inputReady`**`(int fd)`	Called when there is input ready on the file descriptor.
`virtual int` **`outputReady`**`(int fd)`	Called when there is output ready on the file descriptor.
`virtual int` **`exceptionRaised`**`(int fd)`	Called when an exception is raised on the file descriptor.
`virtual void` **`timerExpired`**`(` `long sec, long usec)`	The dpIOHandler timer expired; sec and usec represent the actual time it waited before the timer expired (actual timeout period versus specified time out).

4

Using the Database Query Functions

The database query functions allow a client or provider program to query the provider configuration file. The configuration file is a simple database containing fields that correspond to the key-value pairs in an `XtlKVList`. Each key-value pair describes an aspect of how a provider has been configured on the host. Each query should reference a specific provider alias, which the administrator has defined. See the *Sun XTL 1.1 Administrator's Guide* for more information about provider configuration files and aliases. Table 4-6 shows the database query functions.

Table 4-6 Database Query Functions (from `xtldb.h`)

Function	Description
`extern int xtl_db_verify(void)`	Checks the SunXTL configuration database to ensure that it is valid.
`extern int xtl_provider_names( XtlKVList& names)`	Retrieves a list of all configured provider names and returns it in the `XtlKVList` parameter. Within the `XtlKVList`, the key specifies the provider's secondary alias while the value element contains the primary alias of that provider. Values are of type `XtlString`. The `xtl_provider_names()` function returns a count of provider names retrieved, otherwise it returns `-1` upon error.
`int xtl_provider_info( const XtlString& alias, XtlKVList& info)`	Retrieves all configuration information about the provider specified by `alias` (primary or secondary) and returns the information in an `XtlKVList`. This function returns a count of key-value pairs that were successfully retrieved from the configuration database about the provider, otherwise it returns `-1` upon error.

Code Example 4-3 shows an example of querying the provider configuration database to discover a provider's default speaker and microphone values. The values are then used to configure the call's media channel.

Code Example 4-3 Querying the Database

```
// This code queries the provider configuration database for the default speaker
// and microphone values, and composes a configuration specification to configure
// the media channel through configure_req().

 XtlString input;
 XtlString output;
 XtlKVList defaults;

// pass a provider name and get the provider attributes (keys and values)
```

```
if (xtl_provider_info(call_state(excp).provider()->name(excp), defaults) < 0) {
    fprintf(stderr, "Provider Database not configured?\n");
    return;
}

// find first default microphone key and get its value,
// print error message if not found or not a string.
if (!defaults.first(XtlDBDefaultMicrophoneK) ||  !defaults.get(input)) {
    fprintf(stderr, "Default Input not found.\n");
    return;
}

// find first default speaker key and get its value
if (!defaults.first(XtlDBDefaultSpeakerK) || !defaults.get(output)) {
    fprintf(stderr, "Default Output not found.\n");
    return;
}

// compose the configuration
default_config.add(XtlConfigInputK, input());
default_config.add(XtlConfigOutputK, output());

// This configuration request configures the data stream
// using the key-value pairs contained in the default_config
// argument.
configuration_req(default_config);
```

Using `XtlFormat`

When a client accesses a call's data, it is useful to know the characteristics of that data to properly interpret it—characteristics such as encoding, sample rate, and sample size. SunXTL Teleservices defines several common data formats and characteristics that your program can use. Providers can also define and publish provider-specific data formats for use by the client.

A client can only initialize the data format of a call when it passes the `media_format` parameter to `XtlCall::connect_req()`. After which, the provider associates a format that best matches the requested format. The format that is set can be examined by calling `XtlCallState::format()`. From then on, only the provider can change the format through some provider-specific extension or the provider may be able to determine the data coming over a call and change the format state slot appropriately. For example, an

initial voice call may be directed to fax a message and then return to a human conversation. As the format changes, the provider changes the format slot and informs the client, which can then query the format slot.

A call's data format is described by the `XtlFormat` data type, which is defined as an `XtlKVList`. A format object describes the various characteristics of the call data. A format specification always starts with a *format class* key-value pair (for example, key=`XtlFormatClassK` and value=`XtlVoiceC`), followed by optional key-value pairs that further describe the characteristics of the data. For example, Figure 4-3 shows an `XtlFormat` list that describes an 8 kB sample of μ-law encoded voice data sampled at 8 Hz.

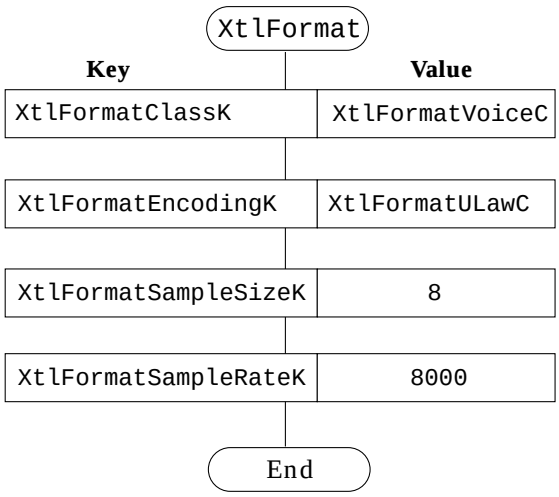


Figure 4-3 Voice Format Specification Example

Table 4-7 shows the predefined format keys and values that programs can use; all types are `XtlString` unless otherwise specified. These well-known keys and values are defined in the `<xtl/constants.h>` header file. If provider-specific keys and values are used, they should be defined in the vendor's provider documentation.

Table 4-7 Predefined `XtlFormat` Keys and Values

Class and Formats	Description
`XtlFormatClassK`	This key is required in all format requests and specifies the general format class. It must be paired with one of the following format values:
`XtlFormatVoiceC`	The media channel contains voice quality audio. This format may specify additional key-value pairs, such as `XtlFormatEncodingK`, `XtlFormatSampleSizeK`, and `XtlFormatSampleRateK`.
`XtlFormatDataC`	Media channel contains uninterpreted data (for example, a raw modem connection). This format contains `XtlFormatBandwidthK`, `XtlFormatFramingK`, and `XtlFormatProtocolK`.
`XtlFormatFaxC`	The media channel contains fax data. This format contains `XtlFormatProtocolK`.
`XtlFormatEncodingK`	Audio data is encoded in one of the following standard formats:
`XtlFormatULawC`	CCITT G.711 μ-law encoding.
`XtlFormatALawC`	CITT G.711 A-law encoding.
`XtlFormatLinearC`	Linear Pulse Code Modulation encoding.
`XtlFormatG721C`	CCITT G.721 compression. This encoding uses Adaptive Delta Pulse Code Modulation with 4-bit precision.
`XtlFormatG723C`	CCITT G.723 compression format. This encoding uses Adaptive Delta Pulse Code Modulation with 3-bit precision.
`XtlFormatSampleSizeK`	Number of bits per sample (stored as an unsigned long in the list).
`XtlFormatSampleRateK`	Rate of sampling flow in samples per second (value is stored as an unsigned long type in the list).
`XtlFormatBandwidthK`	Estimated speed of media channel in bits per second (value is stored as an unsigned long in the list).

Table 4-7 Predefined `XtlFormat` Keys and Values *(Continued)*

Class and Formats	Description
`XtlFormatFramingK`	Data link framing protocol used by a data call.
`XtlFormatHDLC`	HDLC framing.
`XtlFormatProtocolK`	Higher-level protocol used to interpret data.
`XtlFormatIPC`	IP protocol is being sent over a data call.
`XtlFormatG4C`	Group 4 fax protocol is being used for a fax call.
`XtlFormatG3C`	Group 3 fax protocol is being used for a fax call.
`XtlFormatUnknownC`	Use this when a value associated with a particular key is unknown; that is, the protocol is unknown.

Requesting Formats

The underlying technology that a provider uses can support several possible formats for a call. The client can request that a new outgoing call have a particular format by specifying a *format pattern* in the `XtlCall::connect_req()` media format parameter.

The format pattern is an `XtlKVList` that does not have to be a complete format specification; a partial format specification can be passed to the provider. That is, given an incomplete specification, the provider should select a format that most closely satisfies all of the parameters in the format pattern. To do so, the provider uses the `XtlKVList::subset()` method to compare the request with its set of predefined formats. If no matching format can be found, the provider should send an `ERROR_FORMAT_NOT_SUPPORTED` error indication to the call object.

Usage Examples

The following examples show various format request scenarios.

Example 1

Suppose a provider supports both G3 and G4 fax protocols. If the client only wants to make a G3 fax call, it would send the following format pattern:

Key	Value	Key	Value
XtlFormatClassK	XtlFormatFaxC	XtlFormatProtocolK	XtlFormatG3C

where the first key-value pair identifies a fax format class, followed by the specific G3 fax format.

Example 2

A provider supports both μ-law and A-law voice data encodings. The client wants to place a voice call and is capable of handling both μ-law and A-law data. It would send the following format pattern:

Key	Value
XtlFormatClassK	XtlFormatVoiceC

This format specification would match either μ-law or A-law. Once the provider has chosen an encoding, the client examines the format slot in the call object to determine which encoding to use.

Example 3

A provider uses a 2400 bps modem for data communication, but the client wants to place a high-speed data connection. It would send the following format pattern:

Key	Value	Key	Value
XtlFormatClassK	XtlFormatDataC	XtlFormatBandwidthK	9600

Because the provider cannot satisfy the request, it returns `error_ind(ERROR_FORMAT_NOT_SUPPORTED)`.

4

Using `XtlCallReference`

An `XtlCallReference` value is a unique host-wide ID that identifies a call. This ID provides a process-independent handle to a call, thus applications on a host can pass call ownership through any interprocess communication (IPC) mechanism. Having an `XtlCallReference` value also allows your code to identify the `XtlCallState` object associated with a call; with the call state object, clients can then claim or monitor the call.

At the API level, you can obtain a call's reference value by invoking `XtlCallState::call_reference()`. You can then pass the reference value to `XtlProvider::get_call_state_req()` to retrieve the related `XtlCallState` object, or you can pass the reference value to `XtlCallState::provider_name()` to obtain the name of the provider that owns the call.

At the provider level, you obtain a call's reference value by invoking `XtlPCall::get_call_reference()`. You can then pass the reference value to `XtlPProvider::get_call_object()` to obtain a pointer to a call object, or you can pass it to `XtlPFactory::get_provider_name()` to find the name of the call's provider. You can then pass the provider name to `XtlPFactory::get_provider_obj()` to get a pointer to the provider object.

XtlP *Classes* 5

This chapter describes the primary XtlP classes in detail (XtlPFactory, XtlPCall, XtlPProvider, and XtlPPort). Figure 5-1 shows the XtlP class hierarchy. Each of the derived classes contain callback methods. For each of these methods, you must provide code to implement the functionality of your provider.

In addition, there are secondary classes that are used as global data types, such as XtlFormat and XtlCallReference. The information in this chapter serves as a reference for the interfaces to each of these classes. See Chapter 2, “Provider Framework and Concepts” for a description of the relationships between the XtlP classes.

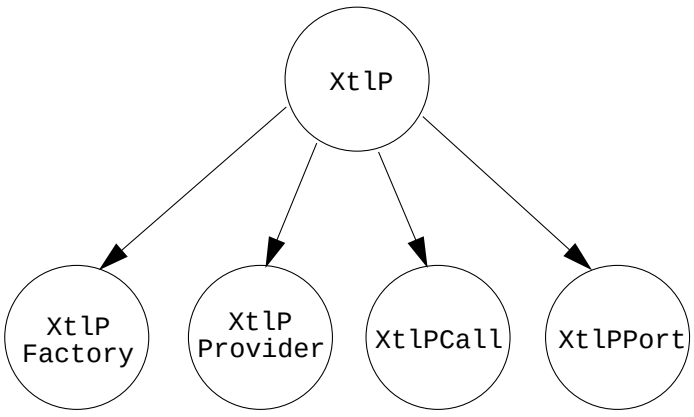


Figure 5-1 XtlP Class Hierarchy

5

`XtlP` *Base Class*

The `XtlP` class is the base class from which the `XtlPFactory`, `XtlPProvider`, `XtlPCall`, and `XtlPPort` classes are derived. Although the `XtlP` class does not contain any interface methods, it does encapsulate enumerated values that define exception, error, and cause codes. Table 5-1, Table 5-2, and Table 5-3 list the `XtlP` class enumerations.

Exception Codes

Exception codes inform the provider of errors resulting from method calls. Each method returns, through its first parameter, one of the exception values shown in Table 5-1. When a method returns, you should check the `Exception` parameter for `EXCEPTION_SUCCESS`; any other value indicates an error condition.

Table 5-1 Exception Codes (from `xtlp_globals.h`)

Exception Codes	Meaning
`EXCEPTION_UNKNOWN`	The exception code was not initialized to any value in this table; this code should never be used.
`EXCEPTION_SUCCESS`	No exception occurred, the operation was successful.
`EXCEPTION_INVALID_FACTORY`	A bad factory object was passed to the provider constructor.
`EXCEPTION_INVALID_PROVIDER`	A bad provider object was passed to the `XtlPCall` constructor. This error indicates an unrecognized provider name or a problem with reading the provider configuration database.
`EXCEPTION_INVALID_ARGUMENT`	An invalid parameter was passed to the method.
`EXCEPTION_INVALID_OBJECT`	The object on which the method was invoked is invalid. This error typically indicates a memory allocation failure, an object initialization error, or a memory corruption error.
`EXCEPTION_INVALID_DATABASE`	The `XtlPFactory` constructor was unable to open the provider configuration database. There may be a problem with the location or permissions of the database.
`EXCEPTION_PROVIDER_NAME_IN_USE`	Returned by the provider constructor when an attempt is made to create a provider with the name of a currently running provider.

Table 5-1 Exception Codes (from `xtlp_globals.h`) *(Continued)*

Exception Codes	Meaning
`EXCEPTION_OUT_OF_MEMORY`	System is out of memory.
`EXCEPTION_PROTOCOL_VIOLATION`	Returned by `event_ind()` when you try to send an event indication while in the wrong state; for example, an invalid state transition occurs if you send a `PROCEEDING_EVENT` while in the `CONNECTED` state. See Table 2-3 on page 18.
`EXCEPTION_INTERNAL_ERROR`	An irrecoverable error has occurred. You should return from the current method and allow the dispatcher to invoke the object's `cleanup()` method.

Error Codes

Error codes are sent to the client as a parameter in the `error_ind()` method. An error indicates an error in a request or an error from the attempt to fulfill a request. Note that `error_ind()` is used only when you need to indicate an error, it does not inform the client of any state changes; you need to use `event_ind()` to inform the client of state changes. Table 5-2 shows the possible error codes.

Table 5-2 Error Codes (from `xtlp_globals.h`)

Error Codes	Meaning
`ERROR_UNKNOWN`	The error code was not initialized to any value in this table; this code should never be used.
`ERROR_INVALID_ARGUMENT`	Bad argument passed in an `XtlKVList`.
`ERROR_INVALID_ADDRESS`	Could not recognize the address.
`ERROR_RESOURCE_NOT_AVAILABLE`	A resource was unavailable while attempting to fulfill a request, such as a configuration or extension request.
`ERROR_RESOURCE_NOT_AVAILABLE_ON_INCOMING`	There was an incoming call, but there were no resources available to accept the call.
`ERROR_PROVIDER_SPECIFIC`	An internal provider-specific error occurred.
`ERROR_PROTOCOL_VIOLATION`	The call is in a state where the attempted request would cause an invalid call state transition; see Figure 2-3 on page 14.
`ERROR_MISSING_PARAMETER`	A parameter is missing.

Table 5-2 Error Codes (from `xtlp_globals.h`) *(Continued)*

Error Codes	Meaning
`ERROR_SERVICE_NOT_IMPLEMENTED`	A request was made for a service that has not been implemented.
`ERROR_SERVICE_NOT_AVAILABLE`	The service requested is unavailable.
`ERROR_SERVICE_NOT_CONFIGURED`	The service was not configured before requesting its use.
`ERROR_NETWORK_NOT_RESPONDING`	The network is ignoring attempts to communicate with it.
`ERROR_FORMAT_NOT_SUPPORTED`	You specified an unsupported format. See “Using XtlFormat” on page 70.
`ERROR_TIMER_EXPIRY`	A timer within the provider timed out. Typically a request may have timed out. For example, a transfer request that gets no response from the destination receives this error code.
`ERROR_REQUEST_CURRENTLY_SATISFIED`	This is a duplicate request for an action that was previously performed. For example, a hold request comes for a call that is already on hold; or, frequently, duplicate configuration requests are sent. In the latter case, it is better to send this error rather than an empty `event_ind(INFO_EVENT)` event because the call state has not changed.

Cause Codes

Cause codes are sent with normal events through the `event_ind()` method. Cause codes provide additional information about an event; for example, a failure event should be accompanied by a cause code to explain why the provider is entering the failed state. Not all events have meaningful causes, so the value `CAUSE_NORMAL` is used in those cases. Table 5-3 lists the possible cause codes.

Table 5-3 Cause Codes (from `xtlp_globals.h`)

Cause Codes	Meaning
`CAUSE_UNKNOWN`	The cause code was not initialized to any value in this table; this code should never be used.
`CAUSE_NORMAL`	This is a normal call control event.
`CAUSE_USER_BUSY`	Remote user terminal was in use.

Table 5-3 Cause Codes (from `xtlp_globals.h`)

Cause Codes	Meaning
`CAUSE_NETWORK_BUSY`	Network was unable to reach the remote party.
`CAUSE_REJECTED`	Call was rejected by the remote party.
`CAUSE_ERROR`	State change was caused by an error.
`CAUSE_ADDRESS_CHANGED`	Provider detected that the remote party's address has changed.

State Slots

State slots hold the attributes of an `XtlP` object and in a call object, they describe the complete state of a call. The `XtlPProvider` and `XtlPCall` classes have a collection of `set_`*slotname*`()` and `get_`*slotname*`()` methods that allow you to set and get the state slots of an object. Slot methods do not take arguments and the *slotname* in the method name corresponds to the name of the state slot. However, certain slots, such as the `call_status` slot, are read-only and therefore do not have a corresponding set method.

Table 5-4 describes the state slots of those classes and shows their default values.This table does not show method arguments, they are shown in the tables for the respective classes (Table 5-7 on page 85 and Table 5-9 on page 88).

Note – There are other method names that begin with `get_`, but are not true slot methods. Rather they take an argument and return a pointer to an object. These non-slot methods include: `XtlPFactory::get_provider_name()`, `get_provider_object()`, `get_provider_list()`, `XtlPProvider::get_call_object()`, and `get_call_list()`.

Table 5-4 State Slots for `XtlP` Classes

State Slot Methods	Default Value	Description
`XtlPProvider`		
`get_provider_name()`	*"primary_alias"*	Returns the provider name (primary alias) of this provider object. Typically, this is the same name as passed to the `XtlPProvider` constructor; however, if a secondary alias was passed, it is resolved to its primary alias.
`set_extended_state()` `get_extended_state()`	`XtlNullKVListC`	Sets and gets the `XtlKVList` that describes the extended state for this provider object.
`XtlPCall`		
`get_call_status()`	`XtlPCall::UNKNOWN`	Returns the `CallStatus` for this call object. The call status is set when this call object sends an event using `XtlCall::event_ind()`.
`get_call_reference()`	`XtlCallReference`	Returns a unique `XtlCallReference` for this call object; the unique value is generated by the MPI.
`get_provider()`	`NULL`	Returns the provider associated with this call object.
`set_local_address()` `get_local_address()`	Zero-length `XtlAddress`	Sets and gets the local address.
`set_remote_address()` `get_remote_address()`	Zero-length `XtlAddress`	Sets and gets the remote address.
`set_format()` `get_format()`	`XtlNullKVListC`	Sets and gets the `XtlFormat`.
`set_media_channel_available()` `get_media_channel_available()`	`B_FALSE`	Sets and gets the availability of the media channel (true or false).
`set_display()` `get_display()`	Null `XtlString`	Sets and gets the `XtlString` shown on the telephone device's display element.
`set_configuration()` `get_configuration()`	`XtlNullKVListC`	Sets and gets the `XtlKVList` that describes the media channel configuration for this call object.
`set_extended_state()` `get_extended_state()`	`XtlNullKVListC`	Sets and gets the `XtlKVList` that describes the extensions used by this call object.

XtlPFactory *Methods*

The XtlPFactory class is used to instantiate provider objects when the client requests a provider. Table 5-5 shows the XtlPFactory class methods, and Table 5-6 shows the class request and event enumerations.

Table 5-5 XtlPFactory Class Methods (from xtlp_factory.h)

Method	How to Use

`XtlPFactory(`
`  Exception&,`
`  const XtlString& provider_family);`
	The object constructor accepts a provider family name as defined in the template file in the provider package.
	
	Possible exceptions (see Table 5-1 on page 78):
	EXCEPTION_OUT_OF_MEMORY
	EXCEPTION_INVALID_DATABASE
	EXCEPTION_INTERNAL_ERROR

`virtual ~XtlPFactory()`	Object destructor.
`virtual void cleanup() = 0;`	This method is invoked when the client no longer wishes to use this provider family. After cleaning up any resources and all providers it has allocated, this method should call exit(). This method must be overridden.

`virtual void create_provider_req(`
`  const XtlString& provider_name,`
`  XtlKVList& args) = 0;`
	Method for instantiating a provider object. This method must be overridden.

`virtual void get_provider_name(`
`  Exception&,`
`  const XtlCallReference&,`
`  XtlString& provider_name);`
	Returns the string name of a provider given the global call ID in the XtlCallReference parameter; see “Using XtlCallReference” on page 75.
	
	Possible exceptions (see Table 5-1 on page 78):
	EXCEPTION_INVALID_ARGUMENT
	EXCEPTION_INVALID_OBJECT

Method	How to Use

`virtual void get_provider_object(`
`  Exception&,`
`  const XtlString&,`
`  XtlPProvider*&);`
	Returns a pointer to a provider object given the string name of the provider.
	
	Possible exceptions (see Table 5-1 on page 78):
	`EXCEPTION_INVALID_ARGUMENT`
	`EXCEPTION_INVALID_PROVIDER`
	`EXCEPTION_INVALID_OBJECT`

`virtual void get_provider_list(`
`  Exception&,`
`  XtlPProvider**& array,`
`  long & size);`
	Returns a list of current provider objects on the host. The list is stored in an array that may be scanned as follows:
	
	`for (int i=0; i < size; i++) {`
	`  myProvider = (myProvider *) array[i];`
	`  // ...operate on the provider object`
	`}`
	
	Possible exceptions (see Table 5-1 on page 78):
	`EXCEPTION_OUT_OF_MEMORY`
	`EXCEPTION_INVALID_OBJECT`

`virtual void error_ind(`
`  Exception&,`
`  Error error,`
`  FactoryRequest providerRequest,`
`  const XtlString& providerName,`
`  const XtlKVList& args);`
	Notification method for reporting errors. `FactoryRequest` is an enumeration with the values: `UNKNOWN_REQ` or `CREATE_PROVIDER_REQ`.
	
	Possible exceptions (see Table 5-1 on page 78):
	`EXCEPTION_INVALID_OBJECT`
	`EXCEPTION_INTERNAL_ERROR`

Table 5-6 `XtlPFactory` Request and Event Codes (from `xtlp_factory.h`)

Provider Event Codes	Description
Requests	
`UNKNOWN_REQ`	The request code was not initialized to any request in this table.
`CREATE_PROVIDER_REQ`	Request to create a new provider.
Events	
`UNKNOWN_EVENT`	The event code was not initialized to any event in this table.
`CREATE_EVENT`	Provider object successfully created.
`INFO_EVENT`	A state slot has changed.

XtlPProvider *Methods*

Table 5-7 presents the interface methods for the XtlPProvider class. Provider objects manage XtlPCall objects, which in turn control individual telephone calls. Table 5-8 shows the request and event enumerations for this class. As a programming note, shaded boxes represent methods that you must implement.

Table 5-7 XtlPProvider Interface Methods (from xtlp_provider.h)

Method	How to Use

`XtlPProvider(`
`  Exception,`
`  XtlFactory&,`
`  const XtlString& provider_name);`
	Constructs a provider object with the specified provider name.
	
	Possible exceptions (see Table 5-1 on page 78):
	EXCEPTION_INVALID_FACTORY
	EXCEPTION_INVALID_PROVIDER
	EXCEPTION_OUT_OF_MEMORY

`virtual ~XtlPProvider();`	Object destructor.
`virtual void cleanup() = 0;`	This cleanup method is called when the client is no longer interested in using this provider. No additional requests will be sent to this provider and sending indication events is considered an error. Your implementation should clean up any provider states, reset your provider hardware, and invoke exit().

`virtual void set_extended_state(`
`  Exception&,`
`  const XtlKVList&);`
	Sets the public provider-specific global state. The state is kept in an XtlKVList that can be passed between the client and the provider.
	
	Possible exception (see Table 5-1 on page 78):
	EXCEPTION_INVALID_OBJECT

`virtual void get_extended_state(`
`  Exception&,`
`  XtlKVList&)`
	Retrieves the provider-specific global state in the list argument.
	
	Possible exception (see Table 5-1 on page 78):
	EXCEPTION_INVALID_OBJECT

`virtual void get_provider_name(`
`  Exception&,`
`  XtlString&)`
	Retrieves the family name of the provider object.
	
	Possible exception (see Table 5-1 on page 78):
	EXCEPTION_INVALID_OBJECT

Table 5-7 `XtlPProvider` Interface Methods (from `xtlp_provider.h`) *(Continued)*

Method	How to Use

`virtual void get_call_object( Exception&, const XtlCallReference& callId, XtlPCall*& call) const;`	Takes the global call ID and retrieves the associated `XtlPCall` object.
	
	Possible exceptions (see Table 5-1 on page 78):
	`EXCEPTION_INVALID_ARGUMENT`
	`EXCEPTION_INVALID_OBJECT`

`virtual void get_call_list( Exception&, XtlPCall**& array, long& size) const;`	Retrieves a list of calls associated with this provider. The list is stored in an array that may be scanned as follows:
	
	`for (int i=0; i < size; i++) {`
	`  myCall= (myCall *) array[i];`
	`  // ...operate on the call object`
	`}`
	
	Possible exceptions (see Table 5-1 on page 78):
	`EXCEPTION_OUT_OF_MEMORY`
	`EXCEPTION_INVALID_OBJECT`

NOTIFICATION METHODS	

`virtual void event_ind( Exception&, XtlPProvider::Eventevent, XtlP::Cause cause, const XtlKVList& args);`	Sends an event indication to the client. Supply a value for the event that occurred (from Table 5-10 on page 94) and a cause of the event (from Table 5-3 on page 80), plus any information you might need to return through the `XtlKVList`.
	
	The only event indication you should send is `CREATE_EVENT`. Valid cause codes that may be sent include: `CAUSE_NORMAL` or `CAUSE_ERROR`.
	
	Possible exceptions (see Table 5-1 on page 78):
	`EXCEPTION_INTERNAL_ERROR`
	`EXCEPTION_INVALID_OBJECT`

`virtual void extension_ind( Exception&, const XtlString&extension, const XtlKVList& args);`	Sends a provider-specific extension indication to the client. The `XtlString` contains the name of the provider-specific feature.
	
	Possible exceptions (see Table 5-1 on page 78):
	`EXCEPTION_INTERNAL_ERROR`
	`EXCEPTION_INVALID_OBJECT`

Table 5-7 `XtlPProvider` Interface Methods (from `xtlp_provider.h`) *(Continued)*

Method	How to Use

`virtual void error_ind(`
`  Exception&,`
`  XtlP::Error  error,`
`  ProviderRequestproviderRequest,`
`  const XtlKVList& args);`
	Sends an error indication to the client. Use an error value from Table 5-2 on page 79 and a provider request value from Table 5-8.
	
	Possible exceptions (see Table 5-1 on page 78):
	`EXCEPTION_INTERNAL_ERROR`
	`EXCEPTION_INVALID_OBJECT`

COMMAND METHOD CALLBACKS	
`virtual void create_call_req(` `  XtlKVList& args) = 0;`	Implementation of behavior when client requests a new call.

`virtual void extension_req(`
`  const XtlString& extension,`
`  XtlKVList& args) = 0;`
	Implementation of behavior when client issues a provider-specific request.

Table 5-8 `XtlPProvider` Request and Event Codes (from `xtlp_provider.h`)

Provider Enumerations	Meaning
Requests	
`UNKNOWN_REQ`	The request value was not initialized to any request in this table; this value should never be used.
`CREATE_CALL_REQ`	Create a call request.
`EXTENSION_REQ`	Provider-specific request.
Events	
`UNKNOWN_EVENT`	The event code was not initialized to any event in this table.
`CREATE_EVENT`	Provider object successfully created.
`INFO_EVENT`	A state slot has changed.

5

`XtlPCall` *Methods*

The `XtlPCall` class encapsulates the components of a call. A call object comes into existence when the provider needs to make a connection and goes away when the provider disconnects a call. Call objects have attributes that may be set and retrieved by using the respective state slot methods: `set_`*slotname*`()` and `get_`*slotname*`()` methods. When a client sets an attribute, it will not see the change until the provider sends the next event. This is so that changes to call objects can be viewed as a single action with respect to events. Table 5-9 presents the interface methods for the `XtlPCall` class.

As a programming note, shaded boxes represent pure virtual functions that you must implement. Also, all `set_`*slotname*`()` and `get_`*slotname*`()` slot methods return either `EXCEPTION_SUCCESS` or `EXCEPTION_INVALID_OBJECT`.

Table 5-9 `XtlPCall` Class Methods (from `xtlp_call.h`)

Method	Description

`XtlPCall(Exception&, XtlPProvider&);`	Call object constructor accepts a reference to the provider that is creating the call.
	
	Possible exceptions (see Table 5-1 on page 78):
	`EXCEPTION_OUT_OF_MEMORY`
	`EXCEPTION_INVALID_PROVIDER`

`virtual ~XtlPCall();`	Call object destructor.
`virtual void cleanup() = 0;`	This cleanup method is called when the client is no longer interested in using this call. No additional requests will be sent to this call and sending indication events is considered an error. Your implementation should perform the necessary provider-specific code to disconnect the call and free up any resources allocated for this call before invoking `exit()`.

Table 5-9 `XtlPCall` Class Methods (from `xtlp_call.h`) *(Continued)*

Method	Description

`virtual void update_port_ind(`
`  Exception&,`
`  XtlPPort* port);`
	Registers a port object with the MPI so that a client may access a call's media channel; see "Presenting Media Channels to the Client" on page 39.
	
	Possible exceptions (see Table 5-1 on page 78):
	`EXCEPTION_INVALID_ARGUMENT`
	`EXCEPTION_INVALID_OBJECT`
	`EXCEPTION_INTERNAL_ERROR`
	An internal error indicates that the MPI was unable to bind the port to the call's media channel; because the error is irrecoverable, the provider should send the client an `error_ind(ERROR_RESOURCE_NOT_AVAILABLE)`.

`virtual void set_local_address(`
`  Exception&,`
`  const XtlAddress& localAddress);`
	Sets the local address from which the call is made.
	Possible exception (see Table 5-1 on page 78):
	`EXCEPTION_INVALID_OBJECT`

`virtual void set_remote_address(`
`  Exception&,`
`  const XtlAddress& remoteAddress);`
	Sets the remote address of the destination to call.

`virtual void set_format(`
`  Exception&,`
`  const XtlFormat & format);`
	Sets the data format used by the call. See "Using XtlFormat" on page 70.

`virtual void`
`set_media_channel_available(`
`  Exception&,`
`  boolean_t mediaChannelAvailable);`
	Turns the media channel on (`B_TRUE`) or off (`B_FALSE`). For information about input/output configurations, see "Configuring Media Channels With configuration_req()" on page 96.

`virtual void set_display(`
`  Exception&,`
`  const XtlString& display);`
	Sets the message to be displayed on the telephone display, such as an LCD display.

`virtual void set_configuration(`
`  Exception&,`
`  const XtlKVList& configuration);`
	Sets the media channel configuration for the call. See "Some Typical Configuration Pairs" on page 98.

`virtual void set_extended_state(`
`  Exception&,`
`  const XtlKVList& extendedState);`
	Sets any provider-specific states using provider-specific key-value pairs.

Table 5-9 `XtlPCall` Class Methods (from `xtlp_call.h`) *(Continued)*

Method	Description
`virtual void get_local_address( Exception&, XtlAddress&);`	Gets the local address of this call.
`virtual void get_remote_address( Exception&, XtlAddress&);`	Gets the remote address of this call.
`virtual void get_format( Exception&, XtlFormat&);`	Gets the data format used by this call.
`virtual void get_media_channel_available( Exception&, boolean_t&);`	Finds out whether the media channel is available for this call.
`virtual void get_display( Exception&, XtlString&);`	Gets the string that is displayed for this call.
`virtual void get_configuration( Exception&, XtlKVList&);`	Gets the current data channel configuration for this call.
`virtual void get_extended_state( Exception&, XtlKVList&);`	Gets the provider-specific state set for this call.
`virtual void get_call_status( Exception&, CallStatus&);`	Finds out what state this call is in. See “Call Status and Transitions” on page 12.
`virtual void get_call_reference( Exception&, XtlCallReference&);`	Gets the reference ID that identifies this call.
`virtual void get_provider( Exception&, XtlPProvider*&);`	Gets the provider that created this call.

Table 5-9 `XtlPCall` Class Methods (from `xtlp_call.h`) *(Continued)*

Method	Description

`virtual void event_ind(`
`  Exception&,`
`  XtlP::Event,`
`  XtlP::Cause,`
`  const XtlKVList& args);`
	Sends an event indication to the client. This informs the client that the state or status of the call has changed. The change may be a change in state or a change in a call attribute such as its quality of service. Any of the events and cause codes can be used; see Table 5-10 on page 94 and Table 5-3 on page 80.
	
	Possible exceptions (see Table 5-1 on page 78):
	`EXCEPTION_PROTOCOL_VIOLATION`
	`EXCEPTION_INVALID_OBJECT`
	`EXCEPTION_INTERNAL_ERROR`

`virtual void extension_ind(`
`  Exception&,`
`  const XtlString& ext_name,`
`  const XtlKVList& args);`
	Sends a provider-specific extension indication to the client to inform it of the results of a provider-specific request.
	
	Possible exceptions (see Table 5-1 on page 78):
	`EXCEPTION_INVALID_OBJECT`
	`EXCEPTION_INTERNAL_ERROR`

`virtual void error_ind(`
`  Exception&,`
`  XtlP::Error,`
`  CallRequest,`
`  const XtlKVList& args);`
	Notifies the client that an error has occurred. You can send any of the codes shown in Table 5-2 on page 79.
	
	Possible exceptions (see Table 5-1 on page 78):
	`EXCEPTION_INVALID_OBJECT`
	`EXCEPTION_INTERNAL_ERROR`

COMMAND METHOD CALLBACKS	

`virtual void connect_req(`
`  const XtlAddress& local,`
`  const XtlAddress& remote,`
`  const XtlFormat&,`
`  XtlKVList&  args) = 0;`
	Use `connect_req()` to establish a connection with the specified remote host. This request is received by the provider only once for a given instance of a call object, and only during the idle state. You need to specify the local address of your device (the caller) and the address of the remote party to be called.
	
	The provider is responsible for selecting among its predefined formats for one that is most comparable to the requested format. This behavior is described in “Using XtlFormat” on page 70.

Table 5-9 `XtlPCall` Class Methods (from `xtlp_call.h`) *(Continued)*

Method	Description
`virtual void add_to_address_req( const XtlAddress& remote, XtlKVList& args) = 0;`	Use this to add to a partial address and to support overlap-send-mode and other address composition features. An `add_to_address_req` is valid only after the `connect_req` has been sent; `add_to_address_req` sends additional addressing information to the provider. Multiple `add_to_address_req` requests may be sent.
`virtual void alert_req( XtlKVList& args) = 0;`	Generates an alert (ring) to inform the remote party that the local client has recognized the incoming call. This request is informational only, thus a provider may generate an alert without first receiving this request. For example, certain technologies (such as ISDN) have a short timeout between an incoming call and a responding alert, after which the call is disconnected. In these situations, your provider may choose to set a timer; if an alert request is not received in that time, it may generate an alert to prevent the call from becoming disconnected. Alert request may be slow in arriving because the client process has been temporarily swapped out of memory.
`virtual void answer_req( XtlKVList& args) = 0;`	Answers an incoming call.
`virtual void disconnect_req( XtlKVList& args) = 0;`	Disconnects a call.
`virtual void hold_req( XtlKVList& args) = 0;`	Requests to release the media channel associated with a call.
`virtual void unhold_req( XtlKVList& args) = 0;`	Requests that the media channel associated with a call be reacquired.
`virtual void transfer_req( const XtlCallReference& transferee, XtlKVList& args) = 0;`	Connects this call to a different call as specified by the `transferee` parameter. See “Transferring a Call” on page 31.
`virtual void conference_req( const XtlCallReference& conferencee, XtlKVList& args) = 0;`	Creates a conference or adds a call to a conference. See “Conferencing a Call” on page 32.
`virtual void redirect_req( const XtlAddress& redirect_number, XtlKVList& args) = 0;`	Sends an incoming call to a different address without answering it.

Table 5-9 `XtlPCall` Class Methods (from `xtlp_call.h`) *(Continued)*

Method	Description
`virtual void drop_req(` `  XtlKVList& args) = 0;`	Removes the last call added to this conference.
`virtual void configuration_req(` `  XtlKVList& args) = 0;`	Configures the media channel. See “Configuring Media Channels With configuration_req()” on page 96.

`virtual void extension_req(`
`  const XtlString& extension,`
`  XtlKVList& args) = 0;`
	Requests a provider-specific feature.

5

Call Event and Request Enumerations

Call request and event enumerations are defined in the `XtlPCall` class; the specific enumerations are shown in Table 5-10 and were also discussed earlier in “Call Status and Transitions” on page 12.

Event codes are defined in the `XtlPCall` and `XtlPProvider` classes and are sent as a parameter in the `event_ind()` method. Event codes inform the client of changes in the call status (see Figure 2-3 on page 14) or state slots (see “Setting and Getting State Slots” on page 11). Most of the event codes correspond to a call status, which should be sent when a call makes a transition from one status to another. However, if the change only affects a state slot, you should use the `INFO_EVENT` code to inform the client. Status changes have precedence over slot changes, so in a case where both call status and slots have changed, you should send the status event.

The provider only generates these events after the fact; that is, after a request or stimuli from the underlying technology causes a change in the call state. There is no guaranteed one-to-one correspondence between a client request and a given event. The basic idea is that events should be treated as independent, asynchronous indications from the provider that the call state has changed. Table 5-10 shows the event codes for the `XtlPCall` class.

Table 5-10 Call Event Codes (from `xtlp_call.h`)

Call Event Codes	Description
`UNKNOWN_EVENT`	The event code was not initialized to any of the values in this table.
`CREATE_EVENT`	Outgoing call created.
`INCOMING_EVENT`	Incoming call created.
`PROCEEDING_EVENT`	The provider has sufficient addressing information to attempt to connect the call.
`ALERTING_EVENT`	Remote party has been notified of the attempt to connect.
`CONNECT_EVENT`	End-to-end connection established.
`FAILURE_EVENT`	Unable to connect because of a failure condition.
`DISCONNECT_EVENT`	Connection terminated
`INFO_EVENT`	A state slot has changed.
`TRANSFER_EVENT`	Call was successfully transferred.

Table 5-10 Call Event Codes (from `xtlp_call.h`) *(Continued)*

Call Event Codes	Description
`CONFERENCE_EVENT`	Call is now part of a conference call.
`REDIRECT_EVENT`	Call has been redirected.
`DROP_EVENT`	Call has been dropped from this conference.

Table 5-11 lists the `XtlPCall` enumerations for call requests. To see the call status and corresponding indication events that are generated when a call makes a transition from one status to another, see Table 2-3 on page 18.

Table 5-11 XtlPCall Class Request Enumerations (from `xtlp_call.h`)

Enumeration	Meaning
`UNKNOWN_REQ`	The call request slot was not initialized with any of the following values:
`CONNECT_REQ`	Request to connect.
`ALERT_REQ`	Request to alert remote part of connection attempt.
`ANSWER_REQ`	Request to answer incoming call.
`DISCONNECT_REQ`	Request to disconnect call.
`HOLD_REQ`	Request to put call on hold.
`UNHOLD_REQ`	Request to take call off hold.
`TRANSFER_REQ`	Request to transfer call to another address.
`CONFERENCE_REQ`	Request to conference this call to another address.
`REDIRECT_REQ`	Request to redirect this call to another address without answering the call.
`DROP_REQ`	Request to drop this call from a conference.
`ADD_TO_ADDRESS_REQ`	Request to add additional addressing information to the call.
`CONFIGURATION_REQ`	Request to configure the media channel.
`EXTENSION_REQ`	Request for provider-specific service.

5

`XtlPPort` *Methods*

When your provider receives a configuration request that asks for client access to a call's media stream, you need to create an `XtlPPort` object to provide that access. The `XtlPPort` class encapsulates a STREAM file descriptor that allows a client to access a call's media stream directly; thus a port object is always associated with a call object. The file descriptor should be opened in read and write modes. The `XtlPPort` class interface consists of a constructor and a destructor:

```
XtlPPort (Exception&, int fd);
~XtlPPort();
```

The constructor accepts an exception output parameter and a file descriptor associated with a STREAM. The possible exceptions that might be returned include: `EXCEPTION_INVALID_ARGUMENT`, `EXCEPTION_OUT_OF_MEMORY`, `EXCEPTION_INTERNAL_ERROR` (see Table 5-1 on page 78 for a description of these exceptions).

Once a port object is created, you associate it with a call by invoking a call object's `update_port_ind()` method. To disassociate a port from a call (and thus disallow the client from accessing the call's media stream), you invoke `update_port_ind(exception, NULL)` with a null port object pointer.

Note – The client synchronizes itself by flushing the port object when it first accesses a call's media stream. For that reason, your provider should make sure any pending data in the port has been used; for example, if your provider writes audio data to a port, allow enough time for the audio to play before invoking `update_port_ind()`.

Configuring Media Channels With `configuration_req()`

The `configuration_req()` method allows the provider to configure the media channel associated with a call. You must override this method before instantiating a call object. The `configuration_req()` method accepts a single argument, an `XtlKVList`, which contains parameters for the desired call configuration.

All call configurations are defined in an `XtlKVList` data type. A key-value pair (KV pair) in an `XtlKVList`, whose key is either `XtlConfigInputK` or `XtlConfigOutputK`, is called a *configuration pair*. If the key for a pair is `XtlConfigInputK`, the configuration element manipulates the input of the media stream. Likewise, if the key for a pair is `XtlConfigOutputK`, the configuration pair manipulates the output of the media stream. The value of a configuration pair is called a *configuration element*. Other KV pairs in the configuration are usually parameters to a configuration pair.

The configuration element whose key is `XtlConfigInputK` can be `XtlConfigStreamC` or a provider-specific value. For example, a provider-specific value for audio media streams might be `XtlDTMFGenerateC`.

The configuration element whose key is `XtlConfigOutputK` may be `XtlConfigStreamC` or a provider-specific value. For example, audio media streams might use the provider-specific value `XtlDTMFDetectC` or `XtlSilenceDetectC`.

To configure and use a provider extension configuration, you would:

1. **Call** `set_configuration()` **with a provider-specific key-value pair.**
   For example, the key `XtlConfigInputK` and the value `XtlDTMFGenerateC`.

2. **Later, to use the provider extension, call** `extension_req()` **with the proper extension name.**
   For example, you might call `extension_req(XtlDTMFToneK, kv_args)`, where `kv_args` would contain the appropriate key-value pairs for generating the tone.

Handling Media Channel Configurations

A `configuration_req()` can be received at any time. If the media stream associated with the call is available at the time the provider receives a `configuration_req()`, the media stream should be configured as requested. The provider then calls `set_configuration()` (passing the configuration in the KVList parameter) and sends an event (most likely `INFO_EVENT`) to inform the MPI of the new configuration.

If the media stream associated with the call is not available at the time a `configuration_req()` is received, the provider should decide how it will attempt to configure the media stream when it does become available. It then

informs the MPI of this configuration by calling `set_configuration()` and sending an event (most likely `INFO_EVENT`). When the data channel becomes available, the provider configures the media stream. If the configuration differs from the one specified in the last `set_configuration()`, the provider must indicate this change in configuration by calling `set_configuration()` and sending an event (most likely `INFO_EVENT`).

Some Typical Configuration Pairs

Each configuration pair specifies how the media stream should be processed. Typically, a processing module (either a software or hardware entity or combination thereof) is used to process the media stream. The `extension_req()` method is invoked when a request is made of a processing module. The status of the data processing is reported to the client through the `extension_ind()` method. Table 5-12 describes the semantics of some common configuration processing pairs that your provider might receive from the client. Your provider may also support additional configurations that you would need to document in your provider package.

Table 5-12 Configuration Pairs for `configuration_req()`

Configuration Pair	Purpose
`<XtlConfigInputK, XtlConfigStreamC>`	This is a configuration request to provide a file descriptor that gives the client write input to the media channel. For a description of how to obtain a stream for a media channel, see “Presenting Media Channels to the Client” on page 39.
`<XtlConfigInputK, XtlDTMFGenerateC>`	This configuration pair requests a DTMF tone generator. When the provider receives an `extension_req` whose extension is `DTMF_GENERATOR`, the `XtlKVList` will contain a KV pair whose key is either `DTMF_TONE` or `DTMF_STRING` (but not both). If the key is `DTMF_TONE`, the value is a `char` cast to a `u_long`. The `char` will be a single character in the set `'[0-9#*ABCD_]'`. The provider then puts the appropriate DTMF tone in the media channel. An underscore character ('_') stops the DTMF tone requested by any previous request; if your device caches tones, it should be flushed. If the `XtlKVList` contains a KV pair with a key `DTMF_STRING`, the `XtlKVList` will contain the following additional KV pairs: **KEY** — **VALUE** `XtlDTMFStringK` — `XtlString (any chars in "[0-9*#ABCD,]*")` `XtlDTMFOffTimeK` — `u_long` (milliseconds) `XtlDTMFOnTimeK` — `u_long` (milliseconds) `XtlDTMFPauseK` — `u_long` (milliseconds)

Table 5-12 Configuration Pairs for `configuration_req()` *(Continued)*

Configuration Pair	Purpose
	This is a request to send a string of DTMF tones with the characteristics given, all durations are in milliseconds. The pause duration specifies how long to pause for any commas (',') in the string; any characters not in the legal set will be silently stripped.
`<XtlConfigOutputK, XtlConfigStreamC>`	This is a configuration request to provide a file descriptor that allows the client to read the output from the media channel. For a description of how to obtain a stream for a media channel, see "Presenting Media Channels to the Client" on page 39.
`<XtlConfigOutputK, XtlDTMFDetectC>`	The `DTMF_DETECTOR` module detects DTMF tones on the data channel. It informs XTLP of DTMF tones by calling `extension_ind()` with the extension set to `DTMF_DETECTOR`, and includes the following KV pair in its `XtlKVList`: **KEY** **VALUE** `XtlDTMFToneK : u_long <char in '[0-9*#ABCD]'>` This indicates a single DTMF character, where the first occurrence of a character indicates the rising edge and the next character indicates a transition ('_' indicates a transition to silence).
`<XtlConfigOutputK, XtlSilenceDetectC>`	The `SILENCE_DETECTOR` module detects the silence or signal on a data channel. If this configuration pair is present in a configuration, the following KV pair must also be present in the configuration: **KEY** **VALUE** `XtlSilenceMinLengthKu_long (milliseconds)` The silence or signal must be present for at least the number of milliseconds specified by the `threshold` parameter to be considered valid. Transitions between silence and signal are indicated by calling `extension_ind()` with extension set to `SILENCE_DETECTOR`. The `XtlKVList` will contain either a KV pair with the key `SILENCE` or `SIGNAL`, depending on whether there is silence or a signal on the line. The type of the value must be a `u_long`, otherwise its value is undefined.

5

Index

Numerics

3-bit precision ADPCM, 72
4-bit precision ADPCM, 72

A

Adaptive Delta Pulse Code Modulation, 72
add(), 58, 59
add_to_address_req(), 92
A-law encoding, 72, 74
alert_req(), 92
alert_req() example, 27
alias, 69
answer_req(), 92
answer_req() example, 27
applications
  Motif, 67
architecture, 2
attribute swapping, 34

B

bits per sample, 72
busy condition, 15
bytes(), 54

C

call
  attribute swapping, 34
  changing the status of, 13
  conferencing, 15, 32
  configuring media channels, 96
  creating, 23
  destroyed, 15
  disconnecting, 28
  dropping from conference, 37
  events, 94
  implicitly register, 24
  operations, 22
  putting off hold, 30
  putting on hold, 30
  requests, 95
  state of, 11, 81
  state vs. status, 12
  status, 12
  status enumerations, 15
  status values, 14
  transferring, 31
  transitions, 12
call progress
  information, 19, 25
  monitoring, 25
call_reference(), 75
callback handler, 4, 19

cause codes, 80
CCITT G.711, 72
class hierarchy, 4
classes
  dpDispatcher, 64
  utility, 53
  XtlByteArray, 53
  XtlKVList, 57
  XtlPCall, 88
  XtlPFactory, 83
  XtlPPort, 96
  XtlPProvider, 85
  XtlString, 55
cleanup(), 83
cleanup() example, 29
client entity, 3
code enumerations, 78
command line environment, 65
comparison function, 61
compiler requirements, 43
compiling the example code, 46
conferee object, 33, 36
conferee parameter, 32
conference call dropping, 37
conference indications, 33
CONFERENCE_EVENT, 33
CONFERENCE_REQ, 17
conference_req(), 92
conference_req() example, 36
conferencing a call, 32
configuration file, 48
configuration pairs, 40, 98
CONFIGURATION_EVENT, 38
configuration_req(), 38, 93, 96
configuring media channels, 38, 96
connect_req(), 73, 91
connecting to the remote party, 15
constants.h, 71
conversion routines, 55
count(), 60
create_call_req(), 87
  example, 23
CREATE_EVENT, 15, 21
create_provider_req(), 20, 21, 83
  example, 21
creating a provider package, 46
creating provider templates, 47
creating providers, 20
creating relocatable packages, 47
current pointer, 58
current position, 58

D

data alignment, 55
data formats, 70
data link framing protocol, 73
database query functions, 69
dependencies in template files, 49
development system requirements, 43
disconnect_req(), 29, 92
disconnecting calls, 28
dispatch loop, 66
dispatch(), 66
dispatch() loop, 27
dispatcher
  initializing, 66
  installing an instance, 67
  loop, 66
dispatcher,*See also* dpDispatcher, 64
dispatcher.h header file, 65
DispatcherMask, 65
documentation, 49
dpDispatcher, 64
  dispatch(), 66
  global dispatcher, 66
  handler(), 65
  instance(), 65, 66
  link(), 65
  setReady(), 66
  startTimer(), 66
  stopTimer(), 66
  unlink(), 65

dpDispatcher methods, 65
dpIOHandler, 64
  exceptionRaised(), 68
  inputReady(), 68
  methods, 68
  outputReady(), 68
  timerExpired(), 68
dpSLDispatcher, 65
dpXtDispatcher, 65
dpXVDispatcher, 65
drop_req(), 37, 93
drop_req() example, 37
dropping a conference call, 37

E

embedded lists, 61
environment
  OLIT, 65
environment variables, 49
environments
  command line, 65
  Motif, 65
  XView, 65
error codes, 79
error indications, 9, 12
ERROR_FORMAT_NOT_
    SUPPORTED, 73, 74
error_ind(), 9, 12, 84, 87, 91
ERROR_REQUEST_CURRENTLY_
    SATISFIED, 12
event dispatcher, 18, 64
event enumerations, 94
event indications, 9, 12
event loop, 45
event_ind(), 9, 12, 25, 86, 91
event-driven programming model, 8
examples
  alert_req(), 27
  answer_req(), 27
  cleanup(), 29
  compiling, 46
  conference_req(), 36
  drop_req(), 37
  formats, 73
  querying the provider database, 69
  template file, 51
  transfer_req(), 31
  XtlByteArray, 55
  XtlString, 56
exception codes, 78
exception on file descriptor, 19
exceptionRaised(), 68
ExceptMask, 65
executable provider, 43
extension indications, 9, 12
extension_ind(), 9, 12, 86, 91
extension_req(), 87, 93

F

fax format class, 74
fax protocols, 74
file descriptor, 19
file descriptor exception, 19
first(), 58, 60
format of template files, 49
formats
  pattern, 73
  requesting, 73
  usage, 73
framework for providers, 7
framing
  HDLC, 73
framing protocol, 73

G

G.721 compression format, 72
G.723 compression format, 72
get methods, 11
get(), 58, 59
get_call_list(), 86
get_call_object(), 75, 86
get_call_reference(), 75, 90
get_call_state_req(), 75

get_call_status(), 90
get_configuration(), 90
get_display(), 90
get_extended_state(), 85, 90
get_format(), 90
get_local_address(), 90
get_media_channel_available(), 90
get_provider(), 90
get_provider_list(), 84
get_provider_name(), 75, 83, 85
get_provider_obj(), 75
get_provider_object(), 84
get_remote_address(), 90
getting state slots, 11
global dispatcher object, 66
Group 3 fax protocol, 73
guaranteeing conference indications, 33

H

handle in the switch, 32
handler for incoming calls, 26
handler(), 65
HDLC framing, 73
header files, 44
hierarchical XtlKVLists, 62
hierarchy, 4
hold_req(), 30, 92
holding calls, 30

I

incoming calls
    handler, 26
    receiving, 26
indication methods, 9, 11
indications
    sending, 12
    valid indication events, 18
INFO_EVENT, 25
initialization
    provider-specific, 21
initialization for providers, 49
initializing the dispatcher, 66
inputReady(), 19, 68
instance(), 65, 66
interface to the client, 3
InterViews dispatcher, 65
InterViews libary, 53
InterViews library, 64
iohandler.h header file, 68
IP protocol, 73

K

key(), 58, 60
key-value pairs, 48, 57

L

lazy copies, 58
LD_LIBRARY_PATH variable, 47, 48
length(), 54, 56
libdispatch library, 64
libdispatch.so, 46
libraries
    libdispatch, 64
libraries of SunXTL 1.1, 46
libxtlp.so, 46
libxtlutil.so, 46
Linear Pulse Code Modulation
        encoding, 72
link(), 18, 65
linking to SunXTL 1.1 libraries, 46

M

main routine, 19
main() loop, 45
masks
    DispatcherMask, 65
    ExceptMask, 65
    ReadMask, 65
    WriteMask, 65
media channels

configuring, 38, 96
presenting to the client, 38
media_channel_available slot, 17
methods of
dpIOHandler, 68
XtlByteArray, 54
XtlKVList, 59
XtlPCall, 88
XtlPFactory, 83
XtlPPort, 96
XtlPProvider, 85
XtlString, 55
m-law encoding, 72, 74
modem communications, 74
monitoring call progress, 25
Motif application, 67
Motif environment, 65
multithread, 4

N

next(), 58, 60

O

OLIT environment, 65
outgoing call
starting, 24
outgoing calls, 15
initiating, 22
outputReady(), 68

P

packaging providers, 43
path of call states, 12
pkgadd(1M), 46
pkginfo(1), 47
primary alias, 69
print(), 61
printing XtlKVList, 62
programming model, 3, 8
protocol violation exception, 16, 18
protocols
Group 3 fax, 73
IP, 73
provider, 8
configuration file, 48
constructor example, 21
creating, 20
creating a package, 46
creating an executable, 43
defined, 7
documentation, 49
family, 8
framework, 7
initialization, 21
initialization script, 49
key-value pairs, 48
names, 69
object, 8
package, 8
packaging, 43
process, 8
programming model, 8
relocatable packages, 47
role of, 1
simulator, 46
template creation at installation, 49
template file example, 51
template file format, 49
templates, 47
writing, 43
provider configuration files, 69
provider_name(), 75
provider-specific extension, 9
provider-specific keys, 47
pure virtual methods, 11

Q

query functions, 69

R

ReadMask, 65
read-only slots, 11, 81
receiving a connection request, 15

receiving requests, 11
redirect_req(), 92
relationships between XtlP classes, 10
relocatable packages, 47
remove(), 58, 59
request, 9
    callback methods, 11
    enumerations, 95
    enumerations call
        request enumerations, 94
    receiving, 11
    valid requests, 17
request methods, 4
requesting a connection, 15
requesting formats, 73
reset(), 58, 60
ringing, 15

S

select(3C), 64, 66
sending indications, 12
set methods, 11
set_configuration(), 39, 89
set_display(), 89
set_extended_state(), 85, 89
set_format(), 89
set_local_address(), 89
set_media_channel_available(), 25, 89
set_remote_address(), 89
setitimer(2), 68
setReady(), 66
setting state slots, 11
simulator provider, 46
startTimer(), 66
state of a call, 11, 81
state slot methods, 88
state slots, 25, 94
    getting, 11
    methods, 11
    read-only, 11, 81
    setting, 11
status enumerations, 15
status of a call, 12
status values, 14
stimuli, 8
stopTimer(), 66
stream head, 40
subset(), 61, 73
SunXTL 1.1 libraries, 43, 46
    linking, 46
SunXTL 1.1 architecture, 2
swapping attributes, 34

T

template file, 47
    creating at installation, 49
    example, 51
    format, 49
timerExpired(), 68
TRANSFER_REQ, 17
transfer_req(), 92
transfer_req() example, 31
transferee parameter, 31
transferring a call, 31
traversing XtlKVLists, 63
type(), 60

U

unhold_req(), 31, 92
unlink(), 65
update_port_ind(), 39, 40, 89, 96
utility classes, 53

V

version value, 49
voice format specification, 71

W

WriteMask, 65
writing a provider program, 43

X

xdr(3N) conversion routine, 55
Xt dispatcher, 67
xtdispatcher.h header file, 65
xtl_db_verify(), 69
xtl_provider_info, 69
xtl_provider_names(), 69
XTL_START key, 47
XtlByteArray
  bytes(), 54
  class, 53
  constructor, 54
  length(), 54
  methods, 54
  usage examples, 55
XtlCall
  connect_req(), 73
XtlCallReference, 31, 32, 75
XtlCallState
  call_reference(), 75
  provider_name(), 75
XtlFormat, 70
XtlFormatALawC, 72
XtlFormatBandwidthK, 72
XtlFormatClassK, 72
XtlFormatDataC, 72
XtlFormatEncodingK, 72
XtlFormatFaxC, 72
XtlFormatFramingK, 73
XtlFormatG3C, 73
XtlFormatG4C, 73
XtlFormatG721C, 72
XtlFormatG723C, 72
XtlFormatHDLCC, 73
XtlFormatIPC, 73
XtlFormatLinearC, 72
XtlFormatProtocolK, 73
XtlFormatSampleRateK, 72
XtlFormatSampleSizeK, 72
XtlFormatULawC, 72
XtlFormatVoiceC, 72
XtlKVList
  add(), 58, 59
  characteristics, 58
  comparison function, 61
  count(), 60
  current pointer, 58
  current position, 58
  first(), 58, 60
  get(), 58, 59
  hierarchical lists, 62
  key(), 58, 60
  next(), 58, 60
  print(), 61
  remove(), 58, 59
  reset(), 58, 60
  subset(), 61, 73
  traversing, 63
  type(), 60
XtlKVList class, 57
XtlKVList methods, 59
XtlP class, 78
XtlP classes, 10
XtlPCall
  get_call_reference(), 75
XtlPCall methods, 88
XtlPCall(), 88
XtlPFactory
  event_ind(), 21
  get_provider_name(), 75
  get_provider_obj(), 75
XtlPFactory methods, 83
XtlPFactory(), 83
XtlPPort, 39
XtlPPort methods, 96
XtlPPort(), 96
XtlPProvider
  get_call_object(), 75
XtlPProvider methods, 85
XtlPProvider(), 85
XtlProvider
  get_call_state_req(), 75
XtlString
  class description, 55

length(), 56
methods, 55
XtlString usage examples, 56
xtltool(1), 46, 48
xv_main_loop(), 67
xvdispatcher.h header file, 65
XView dispatcher, 67
XView environment, 65

Reader Comments

We welcome your comments and suggestions to help improve this manual. Please let us know what you think about the *SunXTL 1.1 Provider Programmer's Guide*, part number *801-7049-11*.

■ The procedures were well documented.

Strongly Agree	Agree	Disagree	Strongly Disagree	Not Applicable
❑	❑	❑	❑	❑

Comments _____

■ The tasks were easy to follow.

Strongly Agree	Agree	Disagree	Strongly Disagree	Not Applicable
❑	❑	❑	❑	❑

Comments _____

■ The illustrations were clear.

Strongly Agree	Agree	Disagree	Strongly Disagree	Not Applicable
❑	❑	❑	❑	❑

Comments _____

■ The information was complete and easy to find.

Strongly Agree	Agree	Disagree	Strongly Disagree	Not Applicable
❑	❑	❑	❑	❑

Comments _____

■ Do you have additional comments about the *SunXTL 1.1 Provider Programmer's Guide*?

_____

_____

_____

_____

Name: _____

Title: _____

Company: _____

Address: _____

_____

Telephone: _____

Email address: _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 1 MOUNTAIN VIEW, CA

POSTAGE WILL BE PAID BY ADDRESSEE

SUN MICROSYSTEMS, INC.
Attn: Manager, Publications
MS MPK 14-101
2550 Garcia Avenue
Mt. View, CA 94043-9850