SunXTL 1.1
Application 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.: 801-7046-11
Revision A, December 1995

Contents

Figures

Tables

Preface

The SunXTL 1.1 Application Programmer's Guide provides information on programming teleservices applications with the SunXTL Application Programmer's Interface (API). The SunXTL API allows you to create applications to make, receive, control, and terminate phone calls in a consistent manner using a variety of telephone switch or network technologies. Note that all references to Solaris in this book apply only to the SPARC version of Solaris.

Audience

This manual is for programmers who have a good understanding of C++ and UNIX, and who wish to develop teleservices applications. You should also be familiar with X-Windows programming concepts such as events, callbacks, and dispatchers.

Purpose of This Manual

This manual describes the classes and methods provided by the SunXTL library. In addition it explains how to use the SunXTL API to write applications that:

- Place or answer multiple calls
- Hold, drop, transfer, and conference calls
- Provide access to media channels
- Enable security and sharing of calls between processes

Structure of This Manual

The chapters in this manual are organized as follows:

Chapter 1, "Introduction," introduces the SunXTL API and SunXTL objects.

Chapter 2, "Getting Started With SunXTL Programming," explains how to compile and run an SunXTL application. It explains configuration and run-time information, and provides programming tips and an example program.

Chapter 3, "Utility Classes," describes the SunXTL utility classes and the event dispatcher classes.

Chapter 4, "SunXTL API Classes," explains the SunXTL programming model. It defines object-oriented programming and describes the SunXTL messaging classes.

Chapter 5, "Creating SunXTL Applications," explains the basic code skeleton necessary for an SunXTL application.

Chapter 6, "Using Media Channels," describes how to access and use media channels between different inputs and outputs.

Appendix A, "Program Examples," provides listings of programs to demonstrate various types of SunXTL applications.

Related Manuals

The SunXTL documentation set also includes these manuals:

- *Sun XTL 1.1 Architecture Guide*
- *Sun XTL 1.1 Administrator's Guide*
- *Sun XTL 1.1 Provider 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`
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 1

The SunXTL application programming interface (API) is a software library that provides connection management and control over telephony and voice services through an object-oriented interface. The API is suitable for implementing telephony applications that make, receive, control, and terminate phone calls.

The SunXTL API is a component of the larger SunXTL platform, which consists of the API, a media platform interface (MPI) library, providers, and configuration databases. The platform is designed to operate with any switch or telephone technology. This platform independence is facilitated by *providers*, which hide the details of the underlying transmission technology; for more information about providers, see the *Sun XTL 1.1 Provider Programmer's Guide*.

Through SunXTL Teleservices, you have access to services that allow your applications to:

- Hold, transfer, conference, and drop telephone calls
- Indicate the progress of a call
- Enable out-of-band call control for automatic call distribution
- Enable audio connections to microphones and speakers
- Detect and generate dual-tone modulated-frequency (DTMF) tones, otherwise known as Touch Tones™

1

The SunXTL API

The SunXTL API uses several important abstractions to represent telephony services and connections. These abstractions are implemented as objects that are created and destroyed with each call. The primary SunXTL classes from which your client applications must derive their classes include:

- `XtlCall`
- `XtlCallState`
- `XtlMonitor`
- `XtlProvider`

Chapter 4, "SunXTL API Classes," describes these classes in detail. There is a loose correspondence between the SunXTL classes and the physical components of a telephone platform: an `XtlProvider` provides access to a telephone technology, an `XtlMonitor` object allows clients to monitor the activities of a call, and `XtlCall` objects with associated `XtlCallState` objects represent the information relevant to a phone call.

The SunXTL platform uses these classes to represent the fundamental elements of a basic telephone platform. These elements are a phone device and the calls made using that device. An `XtlProvider` object represents a valid initialized phone device ready to make and receive calls. It hides the specific device-dependent code driving the telephone device and enables your client application to interact with the telephone device in a device-independent manner.

Your workstation's telephone resources can be shared by many applications. The SunXTL platform controls access to these resources and maintains call data security while enabling cooperating applications to share telephony devices.

SunXTL clients use *media channels* to send and receive data. Media channels can transport various kinds of data such as voice, video, and fax; the type of media and the number of media channels are dependent on the provider. Chapter 6, "Using Media Channels," describes how media channels are used and configured.

SunXTL API Characteristics

The SunXTL API uses an event-driven paradigm where your application is notified of asynchronous events generated by the SunXTL platform. The SunXTL API operates across an asynchronous message-passing interface.

Sending a message to the platform does not guarantee a response. Messages sent from the platform to the application indicate errors and state changes. The cause of a state change is indeterminable—your application will only know that the state has changed.

You should never assume a request will cause a state change, only that it is likely to change to the requested state. Instead, write programs that react to all possible state change messages that are of interest to your client application. Aside from the possible events generated by a network, user input can also cause a program to react. For example, your program should not make the assumption that when a user requests to put a call on hold that the call will be put on hold; the network may have disconnected the call for other reasons.

SunXTL Events

SunXTL applications are driven by events. Each object receives events through its indication methods. Events are messages sent by the provider to notify an application of changes in *call state* or *call progress*. For example, a client can be notified if a call becomes active, on hold, or disconnected.

Event Registration

In order for indication methods to receive events, you must register the events that you are interested in receiving. This is done with the `XtlProvider::listen_req()` method. Later you can choose to disregard certain events by using the `XtlProvider::ignore_req()` method. These methods are described in Table 4-5 on page 52.

SunXTL Object Methods

SunXTL classes provide two types of virtual functions called *request methods* and *indication methods*. These methods generate basic call-control requests that are sent to the provider. Depending on the available telephone resources, the provider may or may not fulfill a request.

Request Methods

Each SunXTL messaging class uses a set of request methods to make service requests to the provider. These methods have names with the suffix `_req`, such as `connect_req()`. An application invokes these methods to manipulate an object. For example, an application program that needs to put a call on hold uses the `XtlCall` object's `hold_req()` request method to change the status of the `XtlCall` object when the call is put on hold.

Indication Methods

Objects receive events from the provider through indication methods. The events notify your application of errors or changes of state on a given call. To handle an event, you must override the indication method of a class. These methods have the suffix `_ind`, such as `event_ind()`. In overriding the indication method, you change the behavior of the class so that it can perform distinct tasks. For example, you can derive the `XtlCall` class to answer a call, play a greeting, and record a message; this behavior imitates an answering machine.

Overview of SunXTL Classes

The SunXTL API features a set of C++ classes to represent various telephony resources. There are two distinct base SunXTL classes: messaging classes and utility classes. These include:

- Messaging Classes: `XtlProvider`, `XtlCall`, and `XtlMonitor`
- Utility Classes: `XtlCallState`, `XtlKVList`, `XtlByteArray`, `XtlString`, `dpIOHandler`, and `dpDispatcher`

Each SunXTL class is an abstraction of a telephony resource. Each class is designed to perform a specific telephony function. For instance, an `XtlProvider` object represents an active telephone device that enables a system to make and receive calls. An `XtlCall` object represents a call in progress, and an `XtlMonitor` object continuously updates and monitors the current state of a call. From these base classes, you can create subclasses to customize their behavior.

Be aware that the messaging classes contain pure virtual functions, which requires implementation to define specific behavior. The utility classes may be instantiated directly. Further information about each class is provided in subsequent chapters, but here is a brief description of each class:

XtlProvider

An `XtlProvider` represents an association with a phone device that enables the system to make and receive calls. You must create provider objects in order to create and receive calls. See “Using XtlProvider” on page 51.

XtlCall

An `XtlCall` represents a call in progress and provides an access method to the call’s data stream. `XtlCall` objects can set up, manipulate, and terminate calls. See “Using XtlCall” on page 56.

XtlMonitor

An `XtlMonitor` enables clients to monitor events on a call. The `XtlMonitor` object allows a client to monitor all call events on a specific call, which it may or may not own.

`XtlMonitor` objects cannot perform call control functions. When an `XtlMonitor` object is created, it is automatically registered to receive all events on a specific call. See “Using XtlMonitor” on page 71.

XtlCallState

The `XtlCallState` class contains state and identification information of a call associated with a specific provider. This information applies to a specific instance of the call only, so an application can use an `XtlCallState` object for immediate status only. `XtlCallState` objects cannot be copied or saved; instead, an `XtlMonitor` object should be used to save the state of the call.

These objects are like handles that enable you to change the ownership of a call and accept incoming calls. All `XtlCallState` object methods return state values only; `XtlCallState` objects do not control calls. See “Using XtlCallState” on page 64.

XtlKVList

An XtlKVList is a singly-linked list of key-value pairs, which are used to pass parameters to the API. Keys are of type XtlString and values may contain various types. The XtlKVList object is often used to pass provider-specific extension information that may not conform to the standard SunXTL interface. See “Using XtlKVList” on page 21.

XtlByteArray

An XtlByteArray is a reference-counted array of bytes. The reference count allows for efficient use of memory because a single XtlByteArray can be shared among a number of objects without concern about memory allocation. The array is deallocated when its reference count is zero. See “Using XtlByteArray” on page 17.

XtlString

An XtlString contains a null-terminated string of characters. Like the XtlByteArray object, it is reference-counted to ease memory management when strings are passed by reference between other objects. See “Using XtlString” on page 19.

dpDispatcher

The dpDispatcher class detects new data on multiple UNIX file descriptors and dispatches the data to the appropriate input and output handlers. See “Using the Event Dispatcher” on page 28.

dpIOHandler

The dpIOHandler class is a simple object that gets called by the dispatcher when data is available on a UNIX file descriptor. If you are familiar with the Interviews dispatcher library, this mechanism of handling I/O is similar. See “Using dpIOHandler” on page 32.

Getting Started With SunXTL Programming 2

This chapter shows you how to compile the SunXTL code example programs provided with the SunXTL package. This chapter also presents the programming concepts and the basic structure used in SunXTL programs, provides some programming tips, and shows an example program that makes outgoing calls.

Compiling SunXTL Applications

Before compiling your applications, be sure that the SUNWxtlb package has been installed (check that the `/opt/SUNWxtl/lib` directory exists). SunXTL applications need to link with two libraries in that directory: `libxtl` and `libxtlutil`.

Note – To use the SunXTL libraries, your development system must be current. For SPARC systems, SunXTL Teleservices requires C++ 4.0.1, available in SPARCompilers 3.0.1.

Creating a Makefile

A makefile greatly automates and eases the compilation and linking process for you. Code Example 2-1 lists the makefile for some of the example programs provided in the SUNWxtl package. The `XTLHOME` variable is set to

the /opt/SUNWxtl directory where the package was installed. From XTLHOME, the location for the library and header files are derived and defined for the INCLUDES and LDFLAGS compiler flags.

Code Example 2-1

```
# Copyright 1993 by Sun Microsystems, Inc.
# @(#)Makefile.demo 1.23


# Parameters.

OPENWINHOME$(OPENWINHOME)=/usr/openwin

#ifdef BUNDLED
XTL_INCLUDES:sh=echo ‘pkginfo -r SUNWxtlh‘/SUNWxtl/include
XTLDEMO:sh=echo ‘pkginfo -r SUNWxtls‘/SUNWxtl/src
XTLLIBDIR:sh=echo ‘pkginfo -r SUNWxtlb‘/usr/xtl/lib
SDK_INCLUDES:sh=echo ‘pkginfo -r SUNWxtls‘/SUNWxtl
#else (!BUNDLED)
XTL_INCLUDES:sh=echo ‘pkginfo -r SUNWxtlh‘/SUNWxtl/include
XTLDEMO:sh=echo ‘pkginfo -r SUNWxtls‘/SUNWxtl/src
XTLLIBDIR:sh=echo ‘pkginfo -r SUNWxtlb‘/SUNWxtl/lib
SDK_INCLUDES:sh=echo ‘pkginfo -r SUNWxtls‘/SUNWxtl/include
#endif (BUNDLED)


# Compiler flags.

INCLUDES += -I$(XTL_INCLUDES) -I$(XTLDEMO) \
     -I/usr/demo/SOUND/include

CPPFLAGS += $(INCLUDES)
LDFLAGS += -L$(XTLDEMO)/datapump  -L$(OPENWINHOME)/lib \
    -L$(XTLLIBDIR) -L/usr/demo/SOUND/lib


LDLIBS += -lxtl -lxtlutil -ldispatch -lnsl \
    -lxview -lolgx -lX11 -lintl -laudio




DEPENDS = \
```

```
# Copyright 1993 by Sun Microsystems, Inc.


.KEEP_STATE:
.INIT: $(DEPENDS)

TARGETS = outcall incall monitorcalls machine detect
all: $(TARGETS)

# program rules

LINK.cc = \
LD_RUN_PATH=$(XTLLIBDIR):$(OPENWINHOME)/lib; export LD_RUN_PATH;



outcall: outcall.o voicecall.o
    $(LINK.cc) -o $@ outcall.o voicecall.o $(LDLIBS)



incall: voicecall.o incall.o
    $(LINK.cc) -o $@ incall.o voicecall.o $(LDLIBS)



monitorcalls: monitorcalls.o
    $(LINK.cc) -o $@ monitorcalls.o $(LDLIBS)



detect: filter.o detect.o
    $(LINK.cc) -o $@ detect.o filter.o $(LDLIBS)



#
# machine program and msgcall.cc use datapump library.  This rule will
# make the library if it does not exist.
#
datapump.lib:
    -@if [ ! -f $(XTLDEMO)/datapump/libdatapump.a ] ; then \
    fi
```

```
# Copyright 1993 by Sun Microsystems, Inc.



machine: machine.o msgcall.o datapump.lib
    $(LINK.cc) -o $@ machine.o msgcall.o -Bstatic -ldatapump
     -Bdynamic $(LDLIBS)


clean:
    -@$(RM) *.o $(TARGETS) *.BAK *.delta
```

How the SunXTL Programming Interface Works

All SunXTL programs use a similar skeleton of code, but differ mainly in the code used for each of the class interface methods. As a minimum, certain classes must be derived to implement the pure virtual functions in a class. The following steps outline the parts of a minimal SunXTL client program. Much of the coding is done in implementing the notification methods and is not shown.

1. **Derive subclasses from the `XtlProvider` and `XtlCall` classes.**
2. **Implement the notification methods in your subclass to receive events.**
3. **To be ready to respond to events, use command methods and create new objects.**

Code Example 2-2 on page 11 uses these steps to create an application that makes an outgoing call.

Some General Concepts

The SunXTL programming paradigm closely follows an X Window programming style. It uses an event model to drive objects and uses a dispatcher (derived from the InterViews™ library) to service those events. Superimpose concepts from telephony and call management and you have the SunXTL programming paradigm. Because we assume you have some X Window programming background, our text concentrates on new concepts introduced by the SunXTL framework.

Program Examples

Before you create new classes, you should consider the kinds of calls your application needs, and the desired behavior for those calls. For example, a program might answer calls in three ways:

- Copy data to an audio device for a human conversation
- Take a message
- Interpret DTMF

In these cases, you can derive three different `XtlCall` classes and implement the three behaviors separately. Then when a provider object gets the incoming call event, it simply creates the call object with the appropriate behavior.

Example Program `outcall.cc`

Code Example 2-2 shows how to make an outgoing call by creating a provider object and call objects. This program creates a provider object that is associated with a device, creates a call object to make calls, registers the provider for incoming call events, and uses a switch statement to act on the events it receives.

For a more detailed explanation of this example program, see “Program Example outcall.cc” on page 75. For now, try compiling the program to verify that your programming environment is set up correctly.

Code Example 2-2 Listing of `outcall.cc`

```
// Copyright 1995 by Sun Microsystems, Inc.



//  outcall
//
//  usage: outcall <remote_number> [provider_name]
//
//  This example creates a single outgoing call to an address
//  specified on the command line.
//

#include <stdio.h>
```

Code Example 2-2 Listing of `outcall.cc` *(Continued)*

```
#include <stdlib.h>

#include <xtl/xtlprovider.h>
#include <xtl/xtlcall.h>
#include <Dispatch/sldispatcher.h>

#include "voicecall.h"

// Class Declarations

// The following classes are derived from XtlProvider and
// DirectAudioCall (defined in voicecall.h) in order to provide
// implementations for the notification callbacks in each class.

//
// MyProvider
//

class MyProvider: public XtlProvider
{
 public:
 MyProvider(Exception*, XtlString, XtlAddress);
 virtual void activated_ind(XtlKVList&);
 virtual void deactivated_ind(XtlKVList&);
 virtual void error_ind(Request, Xtl::Error, XtlKVList&);

 private:
 XtlAddress remote_number;
 class MyCall* current_call;
};

//
// MyCall
//

class MyCall : public DirectAudioCall {
public:
 MyCall(XtlProvider& xtlpv, XtlAddress remoteNumber)
 : DirectAudioCall(xtlpv, remoteNumber) {}
 virtual void deactivated_ind(XtlKVList&);
 virtual void event_ind(CallEvent, XtlKVList&);
};
```

Code Example 2-2 Listing of `outcall.cc` *(Continued)*

```
//
// Class Definitions
//

////////////////////////// MyProvider ////////////////////////

// MyProvider Constructor

MyProvider::MyProvider(
 Exception* err,
 XtlString name,// name of provider to start
 XtlAddress number)// remote number to dial
: XtlProvider(err, name),
 remote_number(number),
 current_call(NULL)
{
}

// The activated method is invoked when the provider is successfully
// initialized. MyProvider::activated creates a new call and attempts
// to dial the number specified on the command line.

void
MyProvider::activated_ind(XtlKVList&)
{
 fprintf(stderr, "Using provider: %s\n", (name())());

 current_call = new MyCall(*this, remote_number);
}

// The deactivated method is invoked when the provider is terminated
// by the system. MyProvider::deactivated exits the program because
// the call will not be completed once the provider has been
// deactivated.

void
MyProvider::deactivated_ind(XtlKVList&)
{
 fprintf(stderr, "Provider died.\n");
 exit(1);
}
```

Code Example 2-2 Listing of `outcall.cc` *(Continued)*

```
// The error method is invoked when the provider detects an error
// condition. MyProvider::error prints the error message and
// the request which caused the error.

void
MyProvider::error_ind(Request req, Xtl::Error err, XtlKVList&)
{
 // convert the request and error values into human readable strings
 XtlStringrequest = XtlProvider::string(req);
 XtlStringerror = Xtl::string(err);

 fprintf(stderr, "Provider error Request %s failed: %s\n",
     request(), error());
}

////////////////////////// MyCall ////////////////////////

// The event method is invoked when "something interesting" happens to
// a call. MyCall::event invokes the default behavior of
// DirectAudioCall to set up the audio device. If the call
// has been disconnected, then MyCall exits the program.

void
MyCall::event_ind(CallEvent event, XtlKVList& kvl)
{
 // preserve DirectAudioCall behavior
 DirectAudioCall::event_ind(event, kvl);

 // exit when call is disconnected
 if (event == DISCONNECT_EVENT)
    exit (0);
}

void
MyCall::deactivated_ind(XtlKVList& kvl)
{
    fprintf(stderr, "Call was destroyed.\n");
    exit (0);
}

////////////////////////// main ////////////////////////

void
```

Code Example 2-2 Listing of `outcall.cc` *(Continued)*

```
main(int argc, char* argv[])
{
 XtlProvider::Exceptionerr;
 XtlAddress address;
 XtlStringprovider_name;
 MyProvider*provider;

 dpDispatcher::instance(new dpSLDispatcher);
 dpDispatcher& d = dpDispatcher::instance();

 // Parse arguments
 if ((argc < 2) || (argc > 3)) {
    fprintf(stderr, "usage: %s <number> [provider]", argv[0]);
    exit(1);
 }

 // save the address and provider name information
 address = XtlByteArray(argv[1]);

 if (argc == 3) {
    provider_name = XtlString(argv[2]);
 }

 // Create a new provider object
 provider = new MyProvider(&err, provider_name, address);

 if (err != MyProvider::EXCEPTION_SUCCESS) {
    fprintf(stderr, "could not connect to provider: %s\n",
     (provider_name() == NULL) ? "default" : provider_name());
    exit(1);
 }

 // enter the dispatch loop
 while(1)
    d.dispatch();
}
```

2

Additional Program Examples

The SunXTL package includes additional program examples you can study to better see how the various classes are used together to perform specific functions such as a dialer, answering machine, or call monitor. These examples are listed in Appendix A, “Program Examples.” The program examples include:

Utility Classes 3

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 3-1 shows the methods provided by the `XtlByteArray` class.

3

Table 3-1 `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 3-1 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 3-1 `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()[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 3-2 shows the methods provided by the `XtlString` class. Code Example 3-2 shows some examples of `XtlString` usage.

Table 3-2 `XtlString` Methods (from `bytearray.h`)

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

Table 3-2 `XtlString` Methods (from `bytearray.h`) *(Continued)*

Method	Description
`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 3-2 `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() {

// XtlString Examples

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

Code Example 3-2 `XtlString` Usage Examples *(Continued)*

```
 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 3-1; 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).

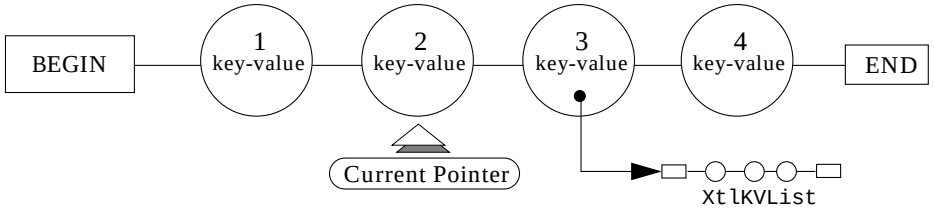

Figure 3-1 `XtlKVList` Structure

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

Table 3-3 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 3-3 `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 3-3 `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 3-3 `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 26 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 kvlist1;

XtlKVList kvlist2;

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

kvlist1.add("key4","val4");
kvlist1.add("key5","val5");

kvlist2 = kvlist1;

kvlist1.add("kvlistkey",kvlist2);
kvlist1.add("key6",6);

kvlist.add("kvlistkey",kvlist1);
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 `XtlKVList`*s*

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 3-2 helps to illustrate these concepts.

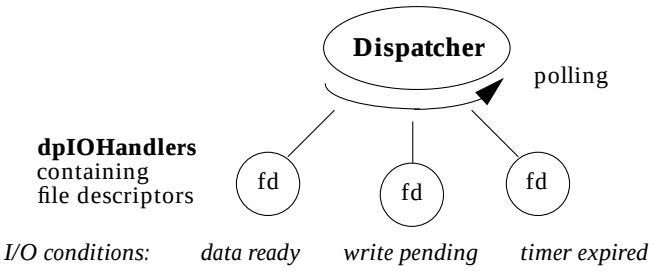

Figure 3-2 `dpDispatcher` and `dpIOHandler` Interactions

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`.

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 3-4 shows the `dpDispatcher` class methods.

Table 3-4 `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.
`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`.

Table 3-4 `dpDispatcher` Methods (from `dispatcher.h`) *(Continued)*

Method	Description
`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 28. 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 3-5 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 3-5 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).

3

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 3-6 shows the database query functions.

Table 3-6 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 3-3 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 3-3 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 3-3 shows an `XtlFormat` list that describes an 8 kB sample of µ-law encoded voice data sampled at 8 Hz.

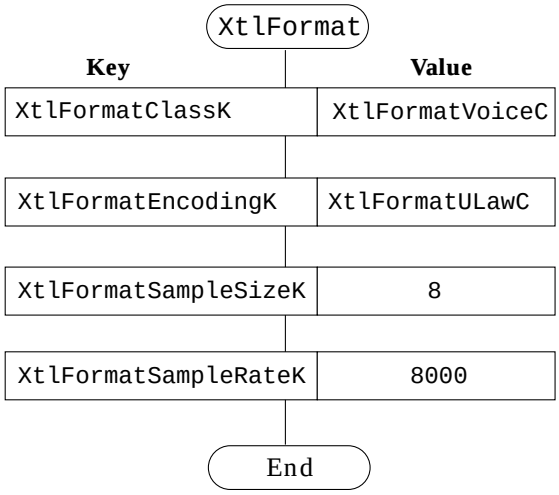

Figure 3-3 Voice Format Specification Example

Table 3-7 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 3-7 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 3-7 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)`.

3

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 `XtlProvider::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.

SunXTL API Classes 4

The SunXTL API classes offer methods that perform basic telephone operations, such as making, receiving, holding, and transferring calls. After an object completes its function, the client can delete it, which frees any resources the object may have used.

You can use this chapter to become familiar with the overall function of each class SunXTL Teleservices provides. For introductory information on using these classes, see “Getting Started With SunXTL Programming” on page 7. For more detailed information on writing specific SunXTL programs see “Creating SunXTL Applications” on page 73. This chapter describes the programming model and methods of each SunXTL class in detail.

Note – The classes and data types in the API library are not multithread safe.

The SunXTL Programming Model

The SunXTL architecture is a message-passing, distributed-object architecture. As such, the model is asynchronous in nature. That is, messages can be sent and received at anytime, objects may come and go at any time. Thus it is important that your programming also be asynchronous in style. Your code should not expect events to occur at a given time or a certain order, and more importantly, your code should not wait or block on events.

The following sections describe the various events that may occur and explain how the SunXTL classes relate and interact.

4

Listening for Events

Because SunXTL clients are event driven, you must decide which events are of interest and then override the indication (callback) methods that will handle the events when they occur. `XtlCall` and `XtlCallState` objects are automatically registered to receive all events when they are constructed; however, `XtlProvider` objects are not registered for any events by default. The `listen_req()` and `ignore_req()` methods allow you to select events of interest.

Note – `XtlProvider` objects cannot receive `CHANNEL_AVAILABLE_EVENT` and `CHANNEL_UNAVAILABLE_EVENT` events directly in the same manner as other SunXTL events. Instead, if the client owns a call, it can listen for these events through its call object. If the client does not own the call, it can create an `XtlMonitor` object to listen for these events. For information about media channels, see Chapter 6, "Using Media Channels."

Note – `listen_req()` cannot be called on `CHANNEL_AVAILABLE_EVENT` or `CHANNEL_UNAVAILABLE_EVENT.`

Events are global enumerated types that indicate a call's change of state. A client can only receive specific events for which it has registered. Events that are not registered will not be sent to the client.

After registering for the events, you need to either override the relevant methods with your own event handler, or do nothing and accept the default behavior. For methods that must be overridden because their default behavior does not adequately handle an event, the `default_method()` method is called automatically, which prints a warning message to `stdout` to indicate that a required method was not overridden.

Event Codes

The `XtlCall` class is the only class that uses events. A client can use an `XtlCall` object to register for events on a particular call, after which, the events are returned on `XtlCall:event_ind()`. Table 4-1 lists the `CallEvent` events. These are events the client may want to listen for and respond to when they occur.

Table 4-1 `CallEvent` Events (from `xtl_globals.h`)

Events	Descriptions
`UNKNOWN_EVENT`	The event parameter was not initialized to any value in this table. A call event should never be set to this value.
Call Progress Events	Call progress events indicate a change in the status of a call. These events can be received by any `XtlCall` object and are guaranteed to be sent when the defined events occur.
`PROCEEDING_EVENT`	An address has been accepted.
`ALERTING_EVENT`	The other end of the call is ringing.
`CONNECT_EVENT`	The call has been connected.
`TRANSFER_EVENT`	A call has been transferred. This event is received by the call object that transferred the call.
`REDIRECT_EVENT`	The call has been redirected.
`CONFERENCE_EVENT`	A call has been connected to a conference.
`DROP_EVENT`	A call has been dropped.
`INFO_EVENT`	The call object has received an information packet from the switch.
`FAILURE_EVENT`	The call has failed to connect.
`DISCONNECT_EVENT`	The call has been disconnected.
Call Ownership Events	Call ownership events occur when a call is first created, changes owner, or becomes invalid (disconnected). These events can only be seen by the `XtlProvider` object, with the exception of the `CHANGE_OWNER_EVENT`, which `XtlMonitor` objects can also receive.
`CHANGE_OWNER_EVENT`	The call now has a new owner. This event causes the associated call object to be deactivated (that is, its `deactivated_ind()` method is invoked).
`CREATE_CALL_EVENT`	A call has been created and its `activated_ind()` method invoked. Only `XtlCall` objects receive this event. Generally, this event implies an outgoing call. This event does not appear for incoming calls where, instead, the client receives an `offer_ind()` and the call state is `INCOMING`.
`INVALIDATE_CALL_EVENT`	A call is no longer valid and the object's `deactivated_ind()` method has been invoked. Only `XtlProvider` objects receive this event.

Table 4-1 `CallEvent` Events (from `xtl_globals.h`) *(Continued)*

Events	Descriptions
Media Channel Events	Calls have associated media channels that contain the media stream (data) of the call. A call must acquire a media channel before it can communicate.
`CHANNEL_AVAILABLE_EVENT`	The media channel associated with the call is available.
`CHANNEL_UNAVAILABLE_EVENT`	The media channel associated with the call is not available.

SunXTL Class Relationships

The SunXTL classes consist of messaging classes (`XtlProvider`, `XtlCall`, and `XtlMonitor`), which share a common base class called `XtlObject`. The remaining utility classes, such as `XtlString` and `XtlKVList`, are stand-alone classes and are used as needed. Some SunXTL objects, however, do interact closely. For instance, `XtlProvider` and `XtlCall` objects are the only objects that can create `XtlCallState` objects. `XtlCallState` objects do not control calls, but contain state information that pertains to a call.

There is also a close relationship between `XtlCallState` objects and `XtlMonitor` objects. An `XtlCallState` object provides information on a specific instance of a call. However, that information is static and provides only a snapshot of a call state in time. The information is static because `XtlCallState` objects cannot send and receive messages. The static nature of the information means you should not copy or save the information. Instead, you must reference the `XtlCallState` through an `XtlMonitor` to retrieve up-to-date call state information.

Initializing Objects

When defining the constructor of a subclass of any of the messaging classes (`XtlProvider`, `XtlCall`, and `XtlMonitor`), you must call the related parent class constructor to initialize the object correctly. If you forget and try to compile a program with an uninitialized parent object, the compile fails with a constructor access error, such as:

```
can’t access private constructor XtlProvider()
```

Interface and Implementation Objects

The SunXTL architecture is based on a message-passing, distributed-object framework that is entirely asynchonous. Thus your programming style must adapt for this environment where messages (requests and indications) may be sent and received at any time, and objects may be destroyed at any time.

Although you are only exposed to the messaging classes, such as `XtlCall` and `XtlProvider`, there are other objects working behind the scene; see Figure 4-1. For each messaging object you instantiate, the API creates a corresponding one, if necessary. The two types of objects are known as *interface objects* and *implementation objects*. Instances of SunXTL messaging classes are referred to as interface objects because they provide a handle, or interface, to corresponding implementation objects, which are hidden. This division allows multiple handles to refer to the same object for efficiency and data sharing purposes. For example, Figure 4-1 shows two interface objects associated (bound) to a single implementation object; the `XtlMonitor` interface object is bound to the same `XtlCall` implementation object in order to monitor the call's activities.

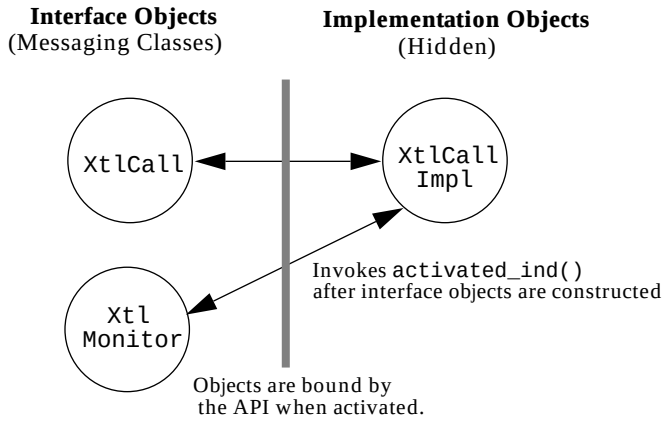

Figure 4-1 Relationship Between Interface and Implementation Objects

When an interface object (also known as a messaging object) is constructed, it must be associated with an implementation object. The API handles this binding automatically through an activation process. Correspondingly, a deactivation process removes the binding when the *implementation object* is destroyed; when the binding is removed, the interface object is no longer valid

and must be destroyed. The activation and deactivation processes are handled by the callbacks `activated_ind()` and `deactivated_ind()`, which must be overridden for each messaging class.

Because implementation objects are hidden, you do not manipulate them directly. But it is helpful to know about them to understand what happens when the API deletes implementation objects; typically, the API invokes the `deactivated_ind()` method to allow you to clean up and delete your interface object; if you do not delete the interface object, any attempt to use it (that is, access the implementation object) results in an `EXCEPTION_INVALID_OBJECT` error.

Activating and Deactivating Messaging Objects

All classes derived from the `XtlObject` class are capable of exchanging messages with a provider to make requests and receive indications—this includes the `XtlProvider`, `XtlCall`, and `XtlMonitor` classes. When you instantiate one of these messaging classes, the object will be activated by the API. This activation occurs automatically after the object is successfully constructed. By activating the object, the object is able to exchange messages with a provider. Likewise, when a provider is done with an object and no longer wants to send and receive messages, it deactivates the object.

The process of activating and deactivating messaging objects is done through two indication methods common to each class: `activated_ind()` and `deactivated_ind()`. These methods are called by the API to notify your client of the state of the object. The API will only invoke these callbacks once in the life of the object. You must override each of these methods.

When you instantiate an object using `new()`, the API automatically registers the object (so that the API can manage it) and invokes the `activated_ind()` method when the object has been registered. Your `activated_ind()` implementation should perform any necessary initialization for the object.

When implementation objects are destroyed, the API automatically unregisters the object and calls the object's `deactivated_ind()` method; within the `deactivated_ind()` method, the object has the opportunity to free resources and clean up any related state information. The object must then destroy itself using `delete()`.

Note – As a rule, the API never deletes memory allocated by an client, nor does it allocate memory and expect the client to delete it. Therefore, it is your responsibility to match `new()` and `delete()` calls for the objects you use.

If you need to delete an object that has not been deactivated by a provider, it is your responsibility to clean up any resources and state information related to the object before calling `delete()`because `delete()` will not call `deactivated_ind()`.

SunXTL Classes

The following SunXTL classes are available:

- Messaging Classes
  - `XtlProvider`
  - `XtlCall`
  - `XtlMonitor`
- Utility Classes (described in Chapter 3, “Utility Classes”)
  - `XtlKVList`
  - `XtlByteArray`
  - `XtlByteString`
  - `XtlCallState`
  - `dpIOHandler`
  - `Dispatcher`

When reading about these methods, notice the correspondence between certain request and indication methods. For example, `listen_req()` and `listen_ind()`, or `get_call_state_req()` and `get_call_state_ind()`, are paired request and indication methods. The suffix indicates the request-indication relationship. For example, when a `listen_req()` request is given, SunXTL Teleservices invokes the `listen_ind()` indication method to return the result and status of the `listen_req()` method. Other methods follow this same convention.

The following sections describe each messaging class's request and indication methods, and the cause, exception, and error codes returned by `event_ind()` and `error_ind()` methods. The utility classes were described in chapter 3.

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 4-2 lists the possible cause codes.

Table 4-2 Cause Codes (from `xtl_globals.h`)

Cause Codes	**Meaning**
`CAUSE_UNKNOWN`	The cause code was not initialized to any of the values 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.
`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.

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 4-3. When a method returns, you should check the `Exception` parameter for `EXCEPTION_SUCCESS`; any other value indicates an error condition.

Table 4-3 Exception Codes (from `xtl_globals.h`)

Exception Codes	Meaning
`EXCEPTION_UNKNOWN`	The exception code was not initialized to any of the values in this table; this code should never be used.
`EXCEPTION_SUCCESS`	No exception occurred, the operation was successful.
`EXCEPTION_INVALID_PROVIDER`	A bad provider object was passed to the call 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 for which the method was called is invalid. This error typically follows a memory allocation failure or indicates that the memory holding the object has been corrupted, thus making the object invalid.
`EXCEPTION_INVALID_DATABASE`	The method was unable to open the configuration database. There may be a problem with the location or permissions of the database.
`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.
`EXCEPTION_INTERNAL_ERROR`	A non-recoverable error has occurred.

4

Error Codes

Error codes are sent 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. Table 4-4 shows the possible error codes.

Table 4-4 Error Codes (from `xtl_globals.h`)

Error Code	Meaning
`ERROR_UNKNOWN = 0`	The error code was not initialized to any of the values in this table; this code should never be used.
`ERROR_INVALID_PROVIDER`	The provider name was not recognized because it was not specified correctly, the provider configuration database is corrupt or missing, or the provider was not installed or configured.
`ERROR_INVALID_OBJECT`	A message was sent to an object that no longer exists. This is an internal error due to a race condition. You may safely ignore this error unless it occurs repeatedly; if so, please report this bug.
`ERROR_INVALID_ARGUMENT`	A bad argument was passed in an `XtlKVList`.
`ERROR_INVALID_ADDRESS`	The provider could not recognize the address.
`ERROR_PERMISSION_DENIED`	A client request required access to a resource for which it did not have permission, such as `/dev/audio`.
`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.
`ERROR_MISSING_PARAMETER`	A parameter is missing.
`ERROR_SERVICE_NOT_IMPLEMENTED`	An unimplemented service was requested.
`ERROR_SERVICE_NOT_AVAILABLE`	The requested service is not available.
`ERROR_SERVICE_NOT_CONFIGURED`	The requested service must be configured (by the system administrator) before the client can use it.
`ERROR_NETWORK_NOT_RESPONDING`	The network is ignoring attempts to communicate with it.

Table 4-4 Error Codes (from `xtl_globals.h`) *(Continued)*

Error Code	Meaning
`ERROR_FORMAT_NOT_SUPPORTED`	You specified an unsupported format. See “Using XtlFormat” on page 35.
`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 will receive this error code.
`ERROR_REQUEST_CURRENTLY_SATISFIED`	This request is a duplicate request for an action that was previously performed. For example, if 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.

Using `XtlProvider`

An `XtlProvider` object represents the service provider that manages connections between a network and a telephone device. `XtlProvider` objects are required for both the creation and reception of connections. Table 4-5 lists the `XtlProvider` class request methods and Table 4-6 shows its slot methods.

The `XtlProvider` constructor accepts an exception parameter, which returns an error code, and an `XtlString` that specifies a provider name. By default, a provider object is not registered to receive any events when it is constructed. You should call `listen_req()` to register for events of interest.

```
XtlProvider* p = new XtlProvider(Exception& exception,
                                 XtlString provider_name);
```

Table 4-5 `XtlProvider` Class Request Methods (from `xtlprovider.h`)

Request Methods	Description
`virtual Exception list_calls_req()`	Requests a list of active calls on this provider.
`virtual Exception enable_offer_event_req( boolean_t on, const XtlKVList& args = XtlNullKVListC)`	Allows the provider object to receive call offer events. To register for offer indications, set the `on` parameter to `B_TRUE`; the optional `args` parameter can be used to specify provider-specific information.
`virtual Exception listen_req( CallEvent listenFor, const XtlKVList& args = XtlNullKVListC)`	By default, provider objects are not registered for any events. Use this method to listen for a named event, specified by `listenFor` (see Table 4-1 on page 43 for list of events). The optional `args` parameter specifies provider-specific information. By design, provider objects cannot listen for `CHANNEL_AVAILABLE_EVENT` and `CHANNEL_UNAVAILABLE_EVENT` events directly in the same manner as other SunXTL events. Instead, if the client owns a call, it can listen for these events through its call object. If the client does not own the call, it can create an `XtlMonitor` object to listen for these events. For information about media channels, see Chapter 6, “Using Media Channels.”
`virtual Exception ignore_req( CallEvent toIgnore, const XtlKVList& args = XtlNullKVListC)`	Unregisters an event, specified by `toIgnore`, with this provider (see Table 4-1 on page 43 for list of events). The optional `args` parameter specifies provider-specific information.
`virtual Exception extension_req( const XtlString& feature, const XtlKVList& args = XtlNullKVListC)`	Requests a provider-specific feature to be activated, such as call forwarding, speed dialing, and so on. You must specify the name of the feature in `feature` and any necessary arguments for the feature in `args`.
`virtual Exception get_call_state_req( const XtlCallReference& ref)`	Gets the state of a call, which is specified by the `XtlCallReference` value.

Table 4-6 `XtlProvider` Class Slot Methods (from `xtlprovider.h`)

Slot Methods	Description
`virtual XtlString name( Exception& exception)`	Returns the primary alias for this provider.
`virtual XtlKVList extended_state( Exception& exception)`	Returns the provider-specific extension state that has been set by the provider.

`XtlString string(Request req)`
`XtlString string(Error error)`
`XtlString string(CallEvent event)`
`XtlString string(CallState state)`
	Converts specified enumeration to an `XtlString`.

Your client program must implement the indication (callback) methods in your derived `XtlProvider` subclass. Your implementation determines the behavior of the provider object when it receives events of interest. You only need to implement the indications for those events. By default, unimplemented indication methods call `default_method()`, which prints a message to warn you that the callback has not been implemented (overridden). Table 4-7 lists the indication methods.

Table 4-7 `XtlProvider` Class Indication Methods (from `xtlprovider.h`)

Indication Methods	Description
`virtual void activated_ind( XtlKVList& args)`	Activates object so that it can receive messages.
`virtual void deactivated_ind( XtlKVList& args)`	Deactivates object so that it can no longer receive messages.
`virtual void list_calls_ind( XtlCallState* const* call_list, int call_count);`	Gets list of active calls. The list is returned in a two-dimensional array of length `call_count`. You may not copy or modify the contents of the array (see “Using XtlCallState” on page 64).
`virtual void enable_offer_event_ind( boolean_t on);`	Confirms the `enable_offer_event_req()` if `on` is returned as `B_TRUE`.
`virtual void listen_ind( CallEvent listeningTo)`	Confirms that the provider is registered to receive the event specified by `listeningTo` (see Table 4-1 on page 43 for list of events).

Table 4-7 `XtlProvider` Class Indication Methods (from `xtlprovider.h`)

Indication Methods	Description
`virtual void ignore_ind( CallEvent ignoring)`	Confirms that the provider will no longer receive the event specified by `ignoring` (see Table 4-1 on page 43 for list of events).
`virtual void get_call_state_ind( XtlCallState& state, XtlKVList& args)`	Returns the call state of a call and any provider-specific arguments.
`virtual void call_event_ind( XtlCallState& callstate, CallEvent event, XtlKVList& args)`	Returns the `event` that has occurred on the call specified by the `callstate` value. Additional provider-specific arguments are given by `args`. When a client receives this indication, the `XtlCallState` argument only provides up-to-date `call_reference()` and `state()` slot information; all other states, such as `media_channel_available()` and `client_state()`, are not set. If a client needs complete state information, it my invoke `XtlProvider::get_call_state_req()` with a call reference value, or create an `XtlMonitor` for the call using the `XtlCallState` value.
`virtual void offer_ind( XtlCallState& offeredCall, XtlKVList& args);`	Notifies the provider that a call `offeredCall` is available to be claimed. Additional provider-specific arguments are given by `args`.
`virtual void info_ind( XtlKVList& args);`	Notifies the provider that the provider state has changed; the `extended_state()` method should be called to examine what has changed.

Table 4-7 `XtlProvider` Class Indication Methods (from `xtlprovider.h`)

Indication Methods	Description
`virtual void error_ind( Request request, Error err, XtlKVList& args)`	An error `err` occurred in a given request `request`. Table 4-8 on page 56 shows the values for `Request` and Table 4-4 on page 50 shows the possible error values. Additional provider-specific arguments are given by `args`.
`virtual void extension_ind( const XtlString& feature, XtlKVList& args)`	Confirms that the provider has received the message sent by `extension_req()`.
`virtual void default_method( const char* methodname)`	This method is called when an indication method has not been overridden. This warning occurs because the default behavior of the method does not appropriately handle an event. When invoked, `default_method()` prints the message: *object_name*`::`*method_name*`() invoked but not overridden, obj=`*this_address*.

The `Request` and `Exception` parameters in Table 4-7 are enumerated types, and can have the values listed in Table 4-8. The `Request` values identify the offending request and correspond to the respective request method names. The `Exception` values explain the type or cause of an error.

Table 4-8 `XtlProvider` Event and Request Values (from `xtlprovider.h`)

Type	Enumerations	Description
enum Event	`UNKNOWN_EVENT = 0`	This value indicates an uninitialized event value. This value should never be used.
	`INFO_EVENT`	This event is sent through `event_ind()` to indicate that one of the slot values has changed.

enum Request	`UNKNOWN_REQ=0`
	`CREATE_REQ`
	`LISTEN_REQ`
	`IGNORE_REQ`
	`LIST_CALLS_REQ`
	`EXTENSION_REQ`
	`GET_CALL_STATE_REQ`
		Request values correspond to the offending request method name. The value `UNKNOWN_REQ` means that the request value was uninitialized; it should never be intentionally set to that value.

Using `XtlCall`

An `XtlCall` object represents a call. The object contains state information about a call and provides access to a call's data through its media channel. Closely associated with `XtlCall` objects are `XtlCallState` objects, which hold state information pertaining to a call; see "Using XtlCallState" on page 64 about the `XtlCallState` class.

The request methods of an `XtlCall` object generate requests and may cause responses for call control functions such as establishing, answering, and releasing calls. They can also transfer, hold, unhold, conference, and drop calls.

The `XtlCall` indication methods (*xxx*`_ind()` callbacks) are called to indicate state changes or may be triggered by requests generated by corresponding request methods (such as `configuration_ind()` is called in response to a `configuration_req()`). You must override the indication methods to implement client-specific behavior.

Transferring Call Ownership

`XtlCall` objects are unique in that they are the only objects that can be owned. The specific client that creates a call object is the sole owner of that call and it alone has permission to manipulate the call. However, in situations where there are several SunXTL clients running, a client may claim a call and decide to pass ownership of the call to another client, the owning client can then offer the call by using `XtlCall::offer_req()`.

There are several things to be aware of when a call is offered:

- A call can be offered while in any state.
- All clients listening for offer events will receive an offer event (through `XtlProvider::offer_ind()` or `XtlMonitor::offer_ind()`), including the client that made the offer. A client can test whether it is already the owner by calling `XtlCallState::owner()`.
- All call state information is passed along with the call, except that any file descriptors associated with its media channel becomes invalid and must be reconfigured using `XtlCall::configuration_req()`; the media channel will have a STREAM-STREAM configuration to show that it is invalid (see “Configuring Channels With File Descriptors” on page 92 for a discussion of media channel configurations). However, all other information, such as client state and provider-specific state information, is passed intact. If provider-specific channel inputs and outputs were previously configured, the provider must determine whether the media channel is still valid when call ownership changes.

When a call is offered, every SunXTL client on the host that is listening for offer events receives an `offer_ind()`. The first to claim the offer takes ownership. This means there is no deterministic way to pass call ownership to a specific client. To overcome this, SunXTL clients must cooperate and agree on a procedure to recognize client-directed offers. The client state slot is one way to specify the recipient of a call offer.

For example, an SunXTL client, *answer_center*, is designed to dispatch calls that may go to either of two other running clients called *info_line* and *emergency_line*. Suppose a call needs to be directed to the *emergency_line* client. The *answer_center* client can specify a key-value pair in the `XtlCallState::client_state()` with a mutually recognized key, such as

"CLIENT_NAME", and a value, which contains the name of the target client, "emergency_line." When a call is claimed, the claiming client should check the client state to check that it is the intended recipient of the call.

Claiming Calls

When a call is offered, any client that is listening for offer events (that is, has invoked `XtlProvider::enable_offer_event_req()`) will receive an offer event indication.

When you claim a call, the state of the call does not change from when it was offered. You can check the state of the call by calling `XtlCallState::state()`. If you need to access the call's media channel (see "Configuring Media Channels" on page 86), remember that the previous owner may have already set a configuration. Thus your code should account for this situation. Three scenarios are possible:

- If the call object's `CallState` is `INCOMING`, the media channel is guaranteed to have a null configuration (`XtlNullKVList`), and you can proceed normally.
- If the call object is in any other state, the media channel configuration is unknown because it may have been previously configured. You can verify that the configuration is satisfactory by calling `XtlCall::configuration()`.

  However, the media channel is invalid if the previous configuration used file descriptors for its input and output. Because file descriptors cannot be passed between clients, the media channel is set to a STREAM-STREAM input/output configuration to indicate that the configuration is invalid.
- You can configure the media channel by calling `XtlCall::configuration_req()`. If you send down a configuration request and the media channel is already configured as requested (this is likely to occur for common configurations such as voice), then your client will receive an `XtlCall::error_ind()` with the error `ERROR_REQUEST_CURRENTLY_SATISFIED`. Your code should handle this appropriately; in this situation, it should ignore the error. One way to avoid the error is to request a null configuration (which effectively resets the channel configuration) before configuring the channel as desired.

Constructing a Call Object

A call object is automatically registered to receive all call events pertaining to the call. The `XtlCall` base class offers the request methods listed in Table 4-9.

Clients can construct an `XtlCall` object in several ways:

- `XtlCall(Exception& exception, XtlProvider& provider, const XtlKVList& args=XtlNullKVListC)`

  This constructor creates a new call object on the specified `provider`; this is how outgoing call objects are created. Provider-specific arguments can be specified in the `args` parameter. Check the `exception` parameter for `EXCEPTION_SUCCESS`.

- `XtlCall(Exception& exception, XtlCallState& callstate)`

  This constructor claims an existing call. The call state can be gotten from `offer_ind()` or a call reference value that was passed from another client (a call state can be derived from a call reference value by using `XtlProvider::get_call_state_req()`). Check the `exception` parameter for `EXCEPTION_SUCCESS`.

- `XtlCall(Exception& exception, XtlCall& call_with_old_behavior)`

  This form of the constructor modifies the behavior of an existing call to follow the behavior of a new call subclass; for example, you might want to take an interactive voice call and modify it to behave as a recorded voice call. Check the `exception` parameter for `EXCEPTION_SUCCESS`.

Table 4-9 `XtlCall` Class Request Methods (from `xtlcall.h`)

Request Methods	Description
`virtual Exception connect_req( const XtlAddress& local, const XtlAddress& remote, const XtlFormat media_format, const XtlKVList& args = XtlNullKVListC);`	Dials a number to create an outgoing call. The `local` parameter specifies the address (or telephone number) of the calling party; `remote` is the called party; `media_format` specifies the desired media channel data format (see "Using XtlFormat" on page 35); and `args` is an optional list of provider-specific information.
`virtual Exception add_to_address_req( const XtlAddress& addition, const XtlKVList& args = XtlNullKVListC);`	Certain providers accept addressing information in parts. This method allows you to append additional addressing information to the initial address given in a previous `connect_req()`. The `addition` parameter specifies a piece of the complete address and `args` is an optional list of provider-specific information.
`virtual Exception answer_req( const XtlKVList& args = XtlNullKVListC);`	Answers, or establishes connection with an incoming call; `args` is an optional list of provider-specific information.
`virtual Exception disconnect_req( const XtlKVList& args = XtlNullKVListC);`	Hangs up a call; `args` is an optional list of provider-specific information.
`virtual Exception hold_req( const XtlKVList& args = XtlNullKVListC);`	Puts a call on hold; `args` is an optional list of provider-specific information.
`virtual Exception unhold_req( const XtlKVList& args = XtlNullKVListC);`	Takes a call off hold; `args` is an optional list of provider-specific information.
`virtual Exception transfer_req( XtlCallState& transfer_call, const XtlKVList& args = XtlNullKVListC);`	Transfers a call by connecting this call to another call, specified by `transfer_call`. The other call must be active (that is, `media_channel_available==B_TRUE`); `args` is an optional list of provider-specific information.

Table 4-9 `XtlCall` Class Request Methods (from `xtlcall.h`) *(Continued)*

Request Methods	Description
`virtual Exception redirect_req( const XtlAddress& redirect_to, const XtlKVList& args = XtlNullKVListC);`	Redirects this call to the specified `redirect_to` address; `args` is an optional list of provider-specific information.
`virtual Exception conference_req( XtlCallState& conferee, const XtlKVList& args = XtlNullKVListC);`	Conferences in another call, specified by `conferee`; `args` is an optional list of provider-specific information.
`virtual Exception drop_req( const XtlKVList& args = XtlNullKVListC);`	Drops last call connected to the conference; `args` is an optional list of provider-specific information.
`virtual Exception offer_req( const XtlKVList& args = XtlNullKVListC);`	Offers this call to other clients and thereby relinquishes ownership of the call if claimed by another client; `args` is an optional list of provider-specific information.
`virtual Exception set_client_state_req( const XtlKVList& client_state = XtlNullKVListC);`	Sets the client state of the call with client-specific information given by `client_state`. See `client_state()` in Table 4-13 on page 64.
`virtual Exception extension_req( const XtlString& feature, const XtlKVList& args = XtlNullKVListC);`	Requests a provider-specific `feature`; `args` is an optional list of provider-specific information.
`virtual int configuration_req( const XtlKVList& requested_cfg)`	Configures or opens a call's data stream with `requested_cfg`. Chapter 6, "Using Media Channels" describes how to specify media channel configurations.
`virtual Exception generate_dtmf_req( XtlString& digits, const XtlKVList& args = XtlNullKVListC);`	Generates one or more DTMF tones. This method is provided for backward compatibility only. To generate DTMF tones, new programs should use the DTMF extension mechanism described in Chapter 6, "Using Media Channels."

Table 4-10 `XtlCall` Class Slot Methods (from `xtlcall.h`)

Slot Methods	Description
`virtual XtlCallState& call_state( Exception& exception)`	Returns reference to `CallState` object.
`virtual XtlKVList& configuration( Exception& exception))`	Returns the current media channel configuration.

`XtlString string(Request req)`
`XtlString string(Error error)`
`XtlString string(CallEvent event)`
`XtlString string(CallState state)`
	Converts the specified enumeration to an `XtlString`.

Clients must override the `XtlCall` indication methods in Table 4-11 to receive call events and errors. These indications all occur asynchornously.

Table 4-11 `XtlCall` Class Indication Methods (from `xtlcall.h`)

Indication Methods	Description
`virtual void activated_ind( XtlKVList& args)`	Indicates the activation of a call object which is then able to receive events; `args` is an optional list of provider-specific information.
`virtual void deactivated_ind( XtlKVList& args)`	Indicates the deactivation of a call object which can no longer receive events; `args` is an optional list of provider-specific information.
`virtual void configuraton_ind( XtlKVList& current_config)`	Indicates a `new configuration indication` and returns the new media channel configuration.
`virtual void event_ind( CallEvent event, Cause cause, XtlKVList& args)`	Indicates arrival of an `event` with a related cause code; `args` is an optional list of provider-specific information.
`virtual void error_ind( Request request, Error error, XtlKVList& args)`	Indicates an error condition with a given request; Table 4-4 on page 50 shows the possible errors values; `args` is an optional list of provider-specific information.

Table 4-11 `XtlCall` Class Indication Methods (from `xtlcall.h`) *(Continued)*

Indication Methods	Description

`virtual void` **`extension_ind`**`(`
`const XtlString& feature,`
`XtlKVList& args)`
	Indicates an `extension indication` and returns the newly activated provider-specific `feature` and its argument `list`.

`virtual void` **`detect_dtmf_ind`**`(`
`char value,`
`XtlKVList&)`
	Indicates a DTMF event. This method is provided for backward compatibility only. To generate DTMF tones, new programs should use the DTMF extension mechanism described in Chapter 6, “Using Media Channels.”

`virtual void` **`default_method`**`(` `const char* method)`	This method is called when an indication method has not been overridden. This warning occurs because the default behavior of the method does not appropriately handle an event. When invoked, `default_method()` prints the message: *object_name*`::`*method_name*`()` `invoked but not overridden,` `obj=`*this_address*.

The `XtlCall` class indication methods include `Request` parameters that are enumerated types, and can have the values listed in Table 4-12:

Table 4-12 `XtlCall` Request Values (from `xtlcall.h`)

Enumeration	Possible Values

Request	`UNKNOWN_REQ = 0`
	`CREATE_REQ`
	`CONNECT_REQ`
	`ANSWER_REQ`
	`DISCONNECT_REQ`
	`HOLD_REQ`
	`UNHOLD_REQ`
	`TRANSFER_REQ,`
	`CONFERENCE_REQ`
	`DROP_REQ`
	`OFFER_REQ`
	`CONFIGURATION_REQ`
	`EXTENSION_REQ`
	`GENERATE_DTMF_REQ`

Using `XtlCallState`

An `XtlCallState` object is different from other objects in that it is not created directly; it does not have a public constructor. Instead it is created by the `XtlProvider` and `XtlCall` objects as needed. `XtlCallState` objects contain read-only state information about a call and the information is retrieved through its state slot methods. `XtlCallState` objects should never be copied or saved because their information is not automatically updated when the state of the call changes. You need to create an `XtlMonitor` object to get continuous updates on a call's state.

Within the client, `XtlCallState` objects are typically used in transfer and conference operations where the second call is passed as an `XtlCallState` object rather than a call object; see Table 4-9 on page 60.

The `XtlCallState` class offers slot methods that return all call-related progress, state, and media channel information. Clients can access this information by calling the `XtlCallState` slot methods listed in Table 4-13.

Table 4-13 `XtlCallState` Slot Methods

Slot Methods	Description
`static XtlString provider_name( XtlCallReference& callref)`	Returns the name of the provider managing this call. The call object is specified by its call reference.
`class XtlProvider& provider()`	Gets reference to provider object.
`CallState state()`	Gets call's progress: connected, alerting, and so on.
`XtlAddress local_address()`	Gets local number.
`XtlAddress remote_address()`	Gets remote number.
`XtlString display()`	Returns the string currently displayed on the phone's display, if any.
`boolean_t media_channel_available()`	Gets status of media channel availability. The availability of the media channel is independent of a call's call progress state. That is, the availability of the media channel may change at any time.
`boolean_t owner()`	True if you own the call.
`boolean_t incoming()`	True if the call is an incoming call.
`boolean_t claimable()`	Finds out if call can be claimed.

Table 4-13 `XtlCallState` Slot Methods *(Continued)*

Slot Methods	Description
`boolean_t held()`	Gets status of hold status. Returns true if the call is currently on hold, which also means that the media channel is not available.
`XtlCallReference call_reference()`	Returns a value that identifies this call. A client can pass this function to another client to exchange information about the call.
`XtlKVList extended_state()`	Returns provider-specific state information about a call. Unlike the client state, which is set by the client, the extended state is set by the provider and is read-only for the client.
`XtlKVList client_state()`	Returns client-specific state information about a call. The client state allows a client to attach client-related information to a call that is not otherwise maintained, such as a time stamp of when a call was initially received, a call's duration, and so on. If a client wishes to pass ownership of a call to another client (through `offer_req()`), the client state is preserved so that it can be examined by the other client.
`XtlFormat format()`	Returns data format of the call media stream.

Call State and Transitions

When making or receiving a call, a call object travels a path of call progress states. These states are defined by `XtlCallState`. The state of a call is important to the client because it acts based on the current state of a call.

To change the state 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.

Figure 4-2 shows some of the possible state 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 call moves from one state to the next, the client receives the appropriate indication event, such as `CREATE_EVENT` or `INCOMING_EVENT`. Table 4-14 on page 68 defines the possible call progress state values.

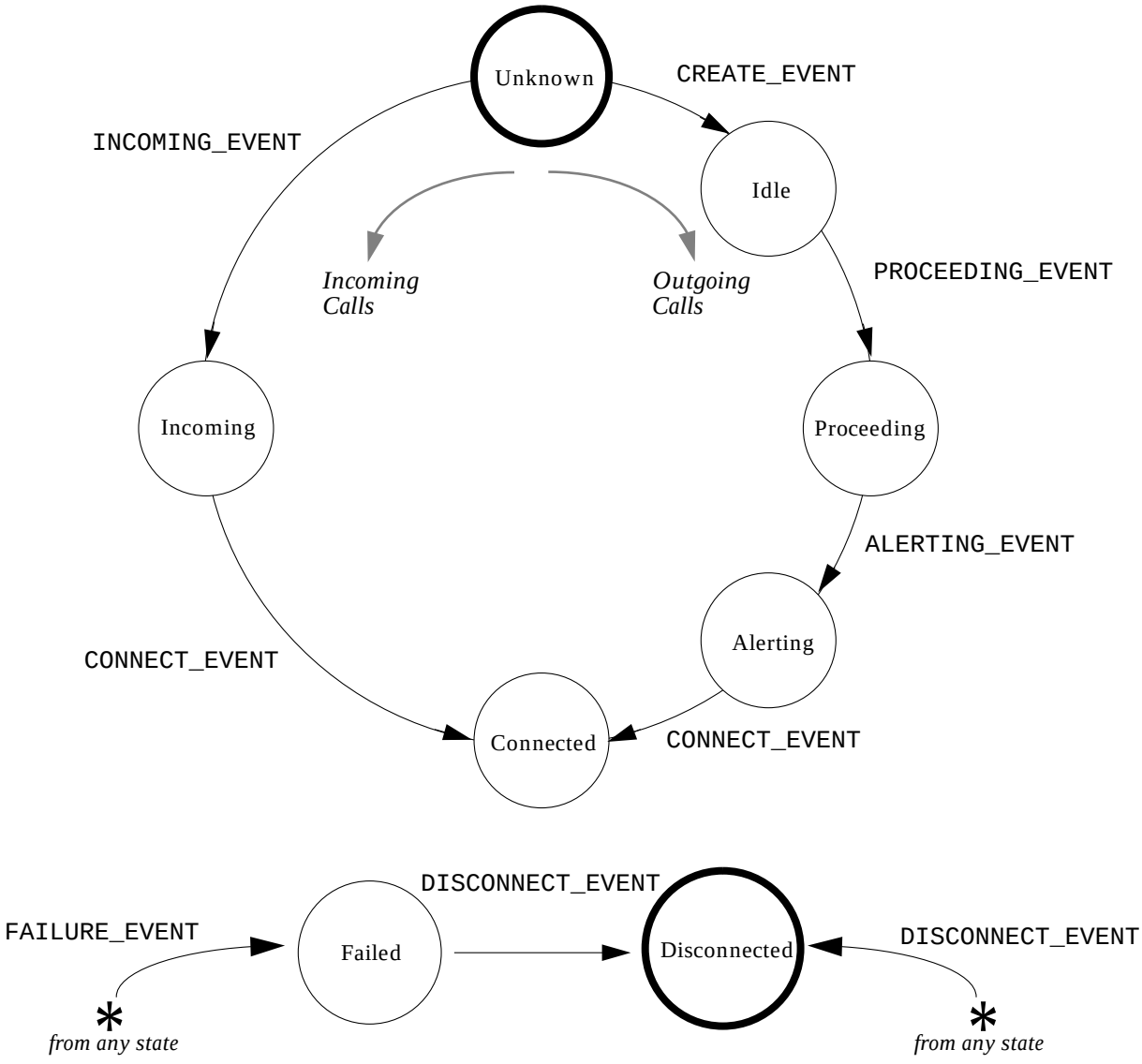

Figure 4-2 Normal Call State Transitions

Table 4-14 Call State Enumerations (from `xtl_globals.h`)

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 may be 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.
`INVALID`	This is not a call state, but rather reflects the state of the `CallState` object. The call may be invalid because the binding to the implementation object has been severed or the object has been deactivated. For a description of object deactivation, see “Activating and Deactivating Messaging Objects” on page 46.

In the normal course of making or receiving a call, a call object can be expected to follow a normal course of state 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 lower-level MPI (provider) validates each transition so that in a given state, only certain requests may be sent to the provider and only certain event indications may be sent to the client.

Table 4-15 shows the valid client requests that a provider can expect to receive for each call state. Table 4-16 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 49.

Table 4-15 Valid Requests for Call States

	Call State[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	-	-	Y	Y	Y	Y	Y	-	Y
CONFERENCE_REQ	-	-	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) A "-" entry in this table means that the MPI guarantees that the request will never be sent in the given state.

Table 4-16 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.

Using XtlMonitor

A monitor object is primarily used to log call activity and monitor the status of incoming calls. It cannot perform call control functions. When a monitor object is created, it is automatically registered to receive all events. Table 4-17 shows the XtlMonitor methods.

The XtlMonitor constructor:

```
XtlMonitor(Exception& exception, XtlCallState& callstate);
```

accepts a `callstate` value, which can be gotten from either `offer_ind()` or a call reference value that was passed from another client (a call state can be derived from a call reference value by using `XtlProvider::get_call_state_req()`). You should also check the `exception` parameter for `EXCEPTION_SUCCESS`.

Table 4-17 `XtlMonitor` Methods

Methods	Description
`virtual XtlCallState& call_state( Exception& exception)`	Returns the call state of the call being monitored. The exception parameter returns any error conditions.
`virtual void activated_ind( XtlKVList& args)`	Activates the monitor object so that it can receive events; `args` is an optional list of provider-specific information.
`virtual void deactivated_ind( XtlKVList& args)`	Deactivates the monitor object so that it no longer receives events; `args` is an optional list of provider-specific information.
`virtual void offer_ind( XtlCallState& offeredCall, XtlKVList& args);`	Notifies the monitor that a call `offeredCall` is available to be claimed. Additional provider-specific arguments are given by `args`.
`virtual void event_ind( CallEvent event, Cause cause, XtlKVList& args);`	Indicates arrival of an `event` with a related cause code; `args` is an optional list of provider-specific information.
`virtual void error_ind( Request request, Error err, XtlKVList& args)`	Sends an error code for a given request. The `Request` value is an enumeration with the values: `UNKNOWN_REQ` and `CREATE_REQ.`
`virtual void default_method( const char* method);`	This method is called when an indication method has not been overridden. This warning occurs because the default behavior of the method does not appropriately handle an event. When invoked, `default_method()` prints the message: *object_name*`::`*method_name*`()` `invoked but not overridden, obj=`*this_address*.

Creating SunXTL Applications 5

The SunXTL API insulates you from the intricacies of the SunXTL platform and the telephone hardware installed on your system. To run the example programs discussed in this chapter, your system should be configured with the appropriate providers for the telephone hardware attached to your system. Consult your system administrator about your system configuration.

This chapter explains how to create and compile SunXTL applications. For a complete description of each SunXTL object, see “SunXTL Classes” on page 47. For introductory information on writing SunXTL applications, see “Getting Started With SunXTL Programming” on page 7.

Creating an SunXTL Application

To create an SunXTL client application requires the following basic steps:

1. **Include SunXTL-specific header files in your source.**
2. **Derive the messaging classes to create your own classes.**
3. **Override the class methods to handle events of interest.**

5

Using Header Files

Several SunXTL header files are available for use in SunXTL applications. To include all the SunXTL header files, simply include the `xtl/xtl.h` header. Some header files are optional, such as `xtl/xtldb.h`, while others, such as `xtl/xtl_globals.h`, are mandatory. Table 5-1 lists the SunXTL header files.

Table 5-1 SunXTL Header Files

Header File	Description
`xtl/xtl.h`	Includes all the headers files shown in this table.
`xtl/bytearray.h`	Defines `XtlByteArray` class.
`xtl/dispatcher.h`	Defines `Dispatcher` class.
`xtl/iohandler.h`	Defines `IOHandler` class.
`xtl/kvlist.h`	Defines `XtlKVList` class.
`xtl/types.h`	Defines simple types such as NULL, nil, true, and false. Also includes the standard <`sys/types.h`> header file.
`xtl/xtl_globals.h`	Defines fundamental SunXTL data types and structures, such as global symbols, event types, and error types.
`xtl/xtlcall.h`	Defines the `XtlCall` class, request types, and request error types.
`xtl/xtlcallstate.h`	Defines `XtlCallState` class.
`xtl/xtldb.h`	Declares database query functions.
`xtl/xtlmonitor.h`	Defines `XtlMonitor` class.
`xtl/xtlprovider.h`	Defines `XtlProvider` class.

Deriving SunXTL Classes

To design your own classes within an application, you derive behavior from the SunXTL messaging classes. For instance, the following code derives a class from the `XtlProvider` class. The new provider subclass, called `MyProvider`, will be activated when its `activated()` method is invoked by the API; that is, it will be activated to send and receive messages.

```
class MyProvider : public XtlProvider {
public:
MyProvider(Exception* err, XtlString pname):XtlProvider(err,
pname) {}
    virtual void activated_ind(XtlKVList&);
    virtual void deactivated_ind(XtlKVList&);
};
```

Because `MyProvider` is derived from `XtlProvider`, `MyProvider` inherits all of the `XtlProvider` class's functionality. The subclass function declarations specify the behavior of the new class, `MyProvider`.

In addition, the `activated_ind()` and `deactivated_ind()` state methods are defined. You must define these methods for every messaging object you derive so that the object is properly initialized when activated and is able to perform any cleanup operations when deactivated.

Note – In order for an application to receive SunXTL events, the class you create to receive that event must derive one of the SunXTL messaging classes: `XtlProvider`, `XtlCall`, or `XtlMonitor`.

Program Example `outcall.cc`

To illustrate the basic code used in an bSunXTL application, the example program `outcall.cc` is explained; Code Example 5-1 presents a listing of the program. The program creates an outgoing call by deriving the SunXTL classes, `MyProvider` and `MyCall` from `XtlProvider` and `DirrectAudioCall`, respectively.

Code Example 5-1 Listing of `outcall.cc`

```
// Copyright 1993 by Sun Microsystems, Inc.
#pragma ident "@(#)outcall.cc1.2493/11/17SMI"



//  outcall
//
//  usage: outcall <remote_number> [provider_name]
//
//  This example creates a single outgoing call to an address
//  specified on the command line.
//

#include <stdio.h>
#include <stdlib.h>

#include <xtl/xtlprovider.h>
#include <xtl/xtlcall.h>
#include <Dispatch/sldispatcher.h>

#include "voicecall.h"

// Class Declarations

// The following classes are derived from XtlProvider and
// DirectAudioCall (defined in voicecall.h) in order to provide
// implementations for the notification callbacks in each class.

//
// MyProvider
//

class MyProvider: public XtlProvider
{
 public:
 MyProvider(Exception*, XtlString, XtlAddress);
 virtual void activated_ind(XtlKVList&);
 virtual void deactivated_ind(XtlKVList&);
 virtual void error_ind(Request, Xtl::Error, XtlKVList&);

 private:
```

Code Example 5-1 Listing of `outcall.cc` *(Continued)*

```
 XtlAddress remote_number;
 class MyCall* current_call;
};

//
// MyCall
//

class MyCall : public DirectAudioCall {
public:
 MyCall(XtlProvider& xtlpv, XtlAddress remoteNumber)
 : DirectAudioCall(xtlpv, remoteNumber) {}
 virtual void deactivated_ind(XtlKVList&);
 virtual void event_ind(CallEvent, XtlKVList&);
};


//
// Class Definitions
//

////////////////////////// MyProvider ////////////////////////

// MyProvider Constructor

MyProvider::MyProvider(
 Exception* err,
 XtlString name,// name of provider to start
 XtlAddress number)// remote number to dial
: XtlProvider(err, name),
 remote_number(number),
 current_call(NULL)
{
}

// The activated method is invoked when the provider is successfully
// initialized. MyProvider::activated creates a new call and attempts
// to dial the number specified on the command line.

void
MyProvider::activated_ind(XtlKVList&)
{
 fprintf(stderr, "Using provider: %s\n", (name())());
```

Code Example 5-1 Listing of `outcall.cc` *(Continued)*

```
 current_call = new MyCall(*this, remote_number);
}

// The deactivated method is invoked when the provider is terminated
// by the system. MyProvider::deactivated exits the program because
// the call will not be completed once the provider has been
// deactivated.

void
MyProvider::deactivated_ind(XtlKVList&)
{
 fprintf(stderr, "Provider died.\n");
 exit(1);
}

// The error method is invoked when the provider detects an error
// condition. MyProvider::error prints the error message and
// the request that caused the error.

void
MyProvider::error_ind(Request req, Xtl::Error err, XtlKVList&)
{
 // convert the request and error values into human readable strings
 XtlStringrequest = XtlProvider::string(req);
 XtlStringerror = Xtl::string(err);

 fprintf(stderr, "Provider error Request %s failed: %s\n",
     request(), error());
}

////////////////////////// MyCall ////////////////////////

// The event method is invoked when "something interesting" happens to
// a call. MyCall::event invokes the default behavior of
// DirectAudioCall in order to set up the audio device. If the call
// has been disconnected, then MyCall exits the program.

void
MyCall::event_ind(CallEvent event, XtlKVList& kvl)
{
 // preserve DirectAudioCall behavior
 DirectAudioCall::event_ind(event, kvl);
```

Code Example 5-1 Listing of `outcall.cc` *(Continued)*

```
 // exit when call is disconnected
 if (event == DISCONNECT_EVENT)
    exit (0);
}

void
MyCall::deactivated_ind(XtlKVList& kvl)
{
    fprintf(stderr, "Call was destroyed.\n");
    exit (0);
}

////////////////////////// main ////////////////////////

void
main(int argc, char* argv[])
{
 XtlProvider::Exceptionerr;
 XtlAddress address;
 XtlString provider_name;
 MyProvider*provider;

 dpDispatcher::instance(new dpSLDispatcher);
 dpDispatcher& d = dpDispatcher::instance();

 // Parse arguments
 if ((argc < 2) || (argc > 3)) {
    fprintf(stderr, "usage: %s <number> [provider]", argv[0]);
    exit(1);
 }

 // save the address and provider name information
 address = XtlByteArray(argv[1]);

 if (argc == 3) {
    provider_name = XtlString(argv[2]);
 }

 // Create a new provider object
 provider = new MyProvider(&err, provider_name, address);

 if (err != MyProvider::EXCEPTION_SUCCESS) {
```

Code Example 5-1 Listing of `outcall.cc` *(Continued)*

```
    fprintf(stderr, "could not connect to provider: %s\n",
     (provider_name() == NULL) ? "default" : provider_name());
    exit(1);
 }

 // enter the dispatch loop
 while(1)
    d.dispatch();
}
```

Most of the SunXTL header files are used in this example with the addition of a custom header file called `voicecall.h`, which defines a class called `DirectAudioCall`. This header file will be used in later examples also.

The first step in the program is to define the `MyProvider` and `MyCall` classes. The `partial listing of the MyProvider` class which follows shows the provider and two event callbacks which signal state changes on the object. The provider object is activated when `activated_ind()` is called and deactivated when `deactivated_ind()` is called.

```
class MyProvider : public XtlProvider {
public:
    MyProvider(Exception* err, XtlString pname) :
                                XtlProvider(err, pname) {}
    virtual void activated_ind(XtlKVList&);
    virtual void deactivated_ind(XtlKVList&);
};
```

The `MyCall` class is derived from the `DirectAudioCall` class, and if you look at the `voicecall.h` file, you can see that `DirectAudioCall` is derived from the `XtlCall` class. Furthermore, `MyCall`'s functionality is extended in

the following listing by defining it to receive call events through `event()`. `MyCall` also implicitly inherits the behavior of the `activated_ind()` and `deactivated_ind()` methods from the `DirectAudioCall` class.

```
class MyCall : public DirectAudioCall {
public:
    MyCall(XtlProvider& xp, XtlAddress rn) :
                              DirectAudioCall(xp, rn) {}
    virtual void            event_ind(CallEvent, XtlKVList&);
};
```

The next step in this program defines an array (`number[80]`) to store the telephone number from the command line argument, and defines a pointer to the current call (`currentCall`).

```
char            number[80];
MyCall*         currentCall=NULL;
```

The following code then defines a `MyProvider` object's response to being activated by `activated_ind()`. This notification is accomplished by overriding the XtlProvider object's `activated_ind()` notification function. The SunXTL platform calls `MyProvider::activated_ind()` to print the provider's primary alias to `stderr`. Then the outgoing call is made by creating the `MyCall` object. This sequence is shown in the following code.

```
void
MyProvider::activated_ind(XtlKVList&)
{
    Exception ex;

    fprintf(stderr, "Using provider: %s\n", name(ex)());

    // Make outgoing call
    currentCall = new MyCall(*this, number);
}
```

Note – The `name()` function always returns the primary alias for the provider even if you specify a secondary alias. Aliases are defined with the `xtltool(1)` configuration utility. For more information on provider aliases, see the *Sun XTL 1.1 Administrator's Guide.*

The following code defines a `MyProvider` object's response to being deactivated.

```
void
MyProvider::deactivated_ind(XtlKVList&)
{
    fprintf(stderr, "Provider died.\n");
    exit(1);
}
```

The `MyCall` event handler uses `DirectAudioCall` to establish an audio connection and calls `exit()` if the connection is disconnected. To customize an application's response to events, you need to override the XtlCall object's `event()` method; in this example, `event()` takes on `DirectAudioCall`'s functionality.

```
void MyCall::event_ind(CallEvent event, Cause, XtlKVList& kvl)
{
    // preserve DirectAudioCall behavior
    DirectAudioCall::event_ind(event, kvl);

    // exit when call is disconnected
    if (event == DISCONNECT_EVENT)
        exit (0);
}
```

Now that everything is defined, the `main()` function is introduced. The argument to `main()` is a provider name. It then sets up the dispatcher and creates a `MyProvider` object. When `MyProvider` is created, and successfully activated, it creates the `MyCall` object to invoke the `connect_req()` call.

```
main(int argc, char* argv[])
{
    XtlProvider::Exceptionerr;
    MyProvider*        Pv;
    Dispatcher&        d = Dispatcher::instance();
    char*              pvname;

    // Parse arguments
    if ((argc < 2) || (argc > 3))  {
        fprintf(stderr, "usage: %s <number> [provider]\n",
                                              argv[0]);
        exit(1);
    }
    strcpy(number, argv[1]);

    if (argc == 3) {
        pvname = argv[2];
    } else {
        pvname = NULL;
    }

    // Connect to provider
    Pv = new MyProvider(&err, XtlString(pvname));
    if (err != XtlProvider::EXCEPTION_SUCCESS) {
        fprintf(stderr, "could not connect to provider\n");
        exit(1);
    }

    while(1)
        d.dispatch();
}
```

5

Using Media Channels 6

In the SunXTL framework, each call has a stream of data associated with it. The data may be voice, fax, video, or some other media. To access this data, a client needs a media channel, which is a configurable port (or handle) to a call's data. This chapter covers concepts concerning SunXTL clients that need to:

- Configure media channels
- Access call data
- Redirect call data between various inputs and outputs

In particular, configuring media channels to access and direct audio data and DTMF information is discussed.

Media channels are automatically allocated for you when you create a call object and deallocated when you destroy the call object. When the media channel is available, the provider sends a `MEDIA_CHANNEL_AVAILABLE` event on the call's `event_ind()` method. It is important to understand that a call's call-progress state is independent of the availability of its media channel. Because media channels are a limited resource, they can be allocated and deallocated at anytime; its availability is notified through `event_ind()` and reflected by the slot method `XtlCall::media_channel_available()`. For example, putting a call in a hold state automatically deallocates the media channel.

6

Configuring Media Channels

To configure media channels, use these methods from the `XtlCall` class:

- `virtual Exception configuration_req(`
  `XtlKVList& new_configuration)`

  This method configures the media channel as described by the values in the `new_configuration` argument. When the media channel is successfully configured, the provider invokes the `configuration_ind`() callback and returns the media channel's current configuration; otherwise the client receives an `error_ind()` event.

- `virtual void configuraton_ind(`
  `XtlKVList& current_configuration)`

  The provider invokes this notification method whenever the media channel configuration changes. Normally the configuration only changes after a `configuration_req()` request has been fulfilled. However, a provider can send a `configuraton_ind()` at any time, so clients must always be prepared to receive this event.

  When a client receives a `configuration_ind()` event, it should compare the returned `current_configuration` with the `new_configuration` that was submitted with the previous `configuration_req()` request. If the configurations are different, you must handle the unexpected configuration appropriately by adapting to the current configuration or trying another `configuration_req()`.

- `virtual XtlKVList& configuration(Exception& exception)`

  This method retrieves the current media channel configuration for your inspection.

Composing a Configuration

A media channel configuration specifies what to connect to the ends of a media channel; that is, its inputs and outputs. To configure a media channel, you need to specify its configuration in an `XtlKVList` and pass it to the `configuration_req()` method. The key-value pairs in the configuration specify the input (data source) and output (data sink) that will be connected to the ends of a media channel. The possible input and output keys and values are defined in the `<xtl/constants.h>` file. Figure 6-1 on page 87 shows the

possible inputs and outputs and how the provider device directs call data through a media channel; there may be several media channels depending on the provider device and each channel can be configured independently.

A configuration is a non-ordered list of inputs and outputs. You can specify multiple outputs, which directs the same data to each output value. Specifying multiple inputs means there are multiple data sources and those sources are merged according to the data type; for example, audio input sources would be mixed.

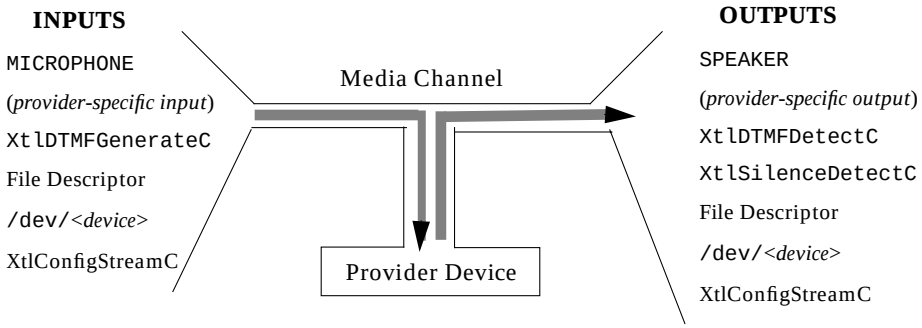

Figure 6-1 Media Channel Input and Output Types

When you submit a configuration through `XtlCall::configuration_req()`, the provider either accepts the request by invoking `configuration_ind()` or it denies the request by invoking `error_ind()`. If the request is accepted, `configuration_ind()` returns the requested configuration as an argument to confirm to request; you might compare this configuration with the one you requested.

A successful configuration request nullifies the previous configuration; that is, you cannot modify parts of a configuration, you must specify a new and complete configuration each time you need to make a configuration change.

Note – A provider may asynchronously change a media channel configuration at any time. This can happen when the provider determines that a call needs to be put on hold or disconnected for resource purposes—for example, to service a higher priority request. The specific behavior or policy adopted by a provider is described in its documentation.

In either situation, the provider should notify the client using `XtlCall::event_ind()` with either `CHANNEL_UNAVAILABLE_EVENT` or `DISCONNECT_EVENT`. These events are equivalent to a `configuraton_ind()` with an empty `XtlKVList` argument. In other words, before the provider dispatches either event, the media channel is set to a null configuration.

Configuring Audio-specific Channels

Media channels are typically configured for audio data. Such configurations can be used for voice calls, playing sound files, and handling DTMF tones. Audio-specific media channels are also flexible in that multiple inputs and outputs can be configured on the channel. The effect is to overlay or mix the sources so that one hears the various audio data simultaneously. Table 6-1 shows the possible input and output key-value pairs for audio channels. You can configure a media channel with any number or combination of these inputs and outputs.

Table 6-1 Audio-specific Key-Value Combinations

Key	Value	Description
`XtlConfigInputK`	`/dev/sound/0`	Directs audio input from the system audio device.
`XtlConfigOutputK`	`/dev/sound/0`	Directs audio output to the system audio device.
`XtlConfigInputK`	*microphone*	Directs audio input from the default microphone; the microphone value should be obtained from the provider configuration file by using `xtl_provider_info(XtlDBDefaultSpeakerK`; for an example, see “Using the Database Query Functions” on page 33.
`XtlConfigOutputK`	*speaker*	Directs audio output to the default speaker; the speaker value should be obtained from the provider configuration file by using `xtl_provider_info(XtlDBDefaultMicrophoneK)`; for an example, see “Using the Database Query Functions” on page 33.
`XtlConfigInputK`	*(provider-specific input)*	Directs audio input from a provider-specific input device. By convention, provider-specific device values should be enclosed in parenthesis; for example, a provider-specific microphone might be called "`(xyz_microphone)`".

Table 6-1 Audio-specific Key-Value Combinations *(Continued)*

Key	Value	Description
`XtlConfigOutputK`	*(provider-specific output)*	Directs audio output to a provider-specific output device. By convention, provider-specific device values should be enclosed in parenthesis; for example, a provider-specific speaker might be called "`(xyz_speaker)`".
`XtlConfigInputK`	`XtlDTMFGenerateC`	Directs input from a DTMF tone generator.
`XtlConfigOutputK`	`XtlDTMFDetectC`	Directs audio output to a DTMF tone detector.
`XtlConfigOutputK`	`XtlSilenceDetectC`	Directs audio output to a DTMF silence detector.

Using DTMF Extensions

Configuring channels to handle DTMF tones is related to audio-specific channel configurations. Composing your configuration request to include DTMF-specific keys and values from the `<xtl/constants.h>` file, together with the `extension_req()` and `extension_ind()` mechanism, allows your client program to perform DTMF tone generation, tone detection, and silence detection.

Because using DTMF tones is an audio operation, you can configure channels with any combination of other inputs and outputs shown previously in Table 6-1 on page 88. Any exceptions should be described in the provider documentation. For example, certain providers cannot perform tone and silence detection simultaneously because a single digital signal processor (DSP) is used to perform both operations.

The following sections describe the steps for using each DTMF configuration.

Generating DTMF Tones

To generate DTMF tones, compose a configuration request that includes the key-value pair:

Key	Value
`XtlConfigInputK`	`XtlDTMFGenerateC`

After a `configuration_ind()` confirms the configuration, generate DTMF tones by invoking:

```
extension_req(XtlDTMFGenerateC, args)
```

where `args` is an `XtlKVList` that contains either a key-value pair whose key is either `XtlDTMFToneK` or `XtlDTMFStringK`, but not both. Table 6-2 describes these keys and their corresponding values.

Table 6-2 DTMF Tone Generation Key-Value Pairs

Key	Value	Description
`XtlDTMFToneK`	`[0-9*#ABCD_]`	Passed as an argument to `extension_req()` to generate a single DTMF tone. The tone plays until the next `extension_req()` is sent with an underbar ('_') value. This key-value pair cannot be used with `XtlDTMFStringK`.
`XtlDTMFStringK`	`[0-9*#ABCD,]*`	Passed as an argument to `extension_req()` to generate a string of DTMF tones; a comma represents a pause. This key-value pair cannot be used with `XtlDTMFToneK`. Several tone attributes (duration, off time, and pause time) can be defined by specifying these additional key-value pairs: **Key** — **Value** `XtlDTMFPauseK` `ulong` (milliseconds) `XtlDTMFOnTimeK` `ulong` (milliseconds) `XtlDTMFOffTimeK` `ulong` (milliseconds) `XtlDTMFPauseK` defines the pause time for a comma (,). `XtlDTMFOnTimeK` defines the duration of each tone. `XtlDTMFOffTimeK` defines the pause between each tone.

DTMF Tone Detection

To detect a DTMF tone, compose a configuration request that includes the key-value pair:

Key	Value
`XtlConfigOutputK`	`XtlDTMFDetectC`

After a `configuration_ind()` confirms the configuration, your program receives an `extension_ind()` callback when a DTMF tone is detected. The `feature` parameter on the callback contains the value `XtlDTMFDetectC`, and the `args` parameter can contain any of the key-value pairs shown in Table 6-2.

DTMF Silence Detection

To detect DTMF silence, compose a configuration request that includes the key-value pair:

KEY	VALUE
`XtlConfigOutputK`	`XtlSilenceDetectC`
`XtlSilenceMinLengthK`	`u_long` (milliseconds)

The `XtlSilenceMinLengthK` key-value pair is optional. It specifies the minimum amount of silence that qualifies as a silence event.

After a `configuration_ind()` confirms the configuration, your program receives an `extension_ind()` callback each time a DTMF signal changes from a tone to silence to a tone; that is, your program receives an event each time a tone appears or silence is detected.

The `feature` parameter on the `extension_ind()` callback contains the value `XtlSilenceDetectC`, and the `args` parameter can contain any of the key-value pairs shown in Table 6-3.

Table 6-3 DTMF Silence Detection Key-Value Pairs

Key	Value	Meaning
`XtlSilenceEventK`	`XtlSilenceEventSilenceC`	Silence was detected on the media channel.
`XtlSilenceEventK`	`XtlSilenceEventSignalC`	A tone was detected on the media channel.

Configuring Channels With File Descriptors

In another variation of channel configuration, you can select channel inputs and outputs that are associated with a file descriptor. These include standard file descriptors, handles to devices, and handles to streams.

In using file descriptors as input and output, you must observe these rules:

- You cannot combine file descriptor input or output with audio-specific input and output.
- You must specify the same file descriptor value for both input and output keys. This is because file descriptors are inherently bidirectional and can serve both input and output functions; that is, a single file descriptor can be read and written with data.
- Unlike audio-specific channel configurations, you can only specify one input and one output per media channel. This is because this data type doesn't allow for logical merging, such as for audio data.
- Any file descriptors returned from a previous `configuraton_ind()` are closed when a configuration is changed.

Note – When you transfer ownership of a call whose media channel is configured with file descriptors, the file descriptors are not carried over. Rather, the file descriptors become invalid and the API resets the configuration to a STREAM-STREAM input/output configuration to indicate that the file descriptors are invalid. The new owner must reconfigure the media channel. See "Transferring Call Ownership" on page 57.

Table 6-4 shows the possible input and output key-value pairs for file descriptors.

Table 6-4 File Descriptor Input and Output Key-value Pairs

Key	Value	Description
`XtlConfigInputK` `XtlConfigOutputK`	<*file descriptor*> <*file descriptor*>	Directs input and output from a file descriptor you have opened. The file descriptor is valid until the next `configuration_ind()`. When a call is on hold, read and write operations will block until the call is taken off hold.
`XtlConfigInputK` `XtlConfigOutputK`	`/dev/`*device* `/dev/`*device*	Directs input and output from a device, where *device* is a device file. The API opens the device for you. The configuration request fails if the device is not available at the time. If you need finer control of resources, you should open the device yourself and pass a file descriptor.

`XtlConfigInputK`
`XtlConfigOutputK`
	`XtlConfigStreamC`
	`XtlConfigStreamC`
		Directs input and output from a stream. The API opens the stream for you and returns a file descriptor. A successful configuration request causes a `configuration_ind()` to return the key-value pairs:
		
		Key — **Value**
		`XtlConfigInputK` — `XtlConfigStreamC`
		`XtlConfigOutputK` — `XtlConfigStreamC`
		`XtlConfigInputFdK` — `<file descriptor>`
		`XtlConfigOutputFdK` — `<file descriptor>`

6

Program Examples A

This chapter lists the program examples provided with the SunXTL software package. To compile these programs, your development system must be current. For SPARC systems, use C++ 3.0.1, in SPARCompilers 2.0.1. For Intel systems, use C++ 3.0.1, in ProWorks 2.0.1.

When installed, the program example files can be found under `/opt/SUNWxtl/src/examples`. There is also a makefile that compiles each of the programs for you:

```
$ cd /opt/SUNWxtl/src/examples
$ make
```

Handling Audio

Code Example A-1 and Code Example A-2 show an example of an SunXTL audio implementation. This program's header file contains a useful data structure, `DirectAudioCall`, which is also used in the other SunXTL programming examples.

Code Example A-1 Listing of `voicecall.h`

```
/* Copyright 1993 by Sun Microsystems, Inc. */
#pragma ident "@(#)voicecall.h1.1994/02/08SMI"

#ifndef _XTL_VOICECALL_H
#define _XTL_VOICECALL_H

// Copyright (c) 1992 by Sun Microsystems, Inc.
```

Code Example A-1 Listing of `voicecall.h` *(Continued)*

```
/* Copyright 1993 by Sun Microsystems, Inc. */
#pragma ident "@(#)voicecall.h1.1994/02/08SMI"

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

//
// DirectAudioCall is an example of a higher level, easier to
// use XtlCall object.
//
// The primary function of DirectAudioCall is to automatically connect a
// calls data stream to the provider's default microphone and speaker.
//
// The DirectAudioCall outgoing call constructor will also automatically
// dial the outgoing number after being created.
//
// The DirectAudioCall claim call constructor claims on "offered" call.
// If the call is an incoming call, the object will automatically
// answer the incoming call.
// If the call is in another state the object will correctly configure
// the data stream of the call.
//
// The default implementation of DirectAudioCall configures the data
// stream using default values in the provider data base. This behavior
// is easily modified by subclassing DirectAudioCall and overriding
// send_desired_config() so it will send the desired configuration.
//
// DirectAudioCall also prints asynchronous errors to stderr.
//
class DirectAudioCall : public XtlCall {
public:
    DirectAudioCall(// Outgoing Call constructor
     Xtl::Exception& err,
     XtlProvider& xtlpv,
     XtlAddress& remoteNumber)
     : XtlCall(err, xtlpv), _remoteNumber(remoteNumber) {}
    DirectAudioCall(// Claim Call constructor
     Xtl::Exception& err,
     XtlCallState& call)
     : XtlCall(err, call), _remoteNumber("") {}
    virtual void activated_ind(XtlKVList&);
    virtual void event_ind(CallEvent, Cause, XtlKVList&);
    virtual voidsend_desired_config();
    virtual void extension_ind(const XtlString&, XtlKVList&);
```

Code Example A-1 Listing of `voicecall.h` *(Continued)*

```
/* Copyright 1993 by Sun Microsystems, Inc. */
    virtual voidconfiguration_ind(XtlKVList&);
    virtual void error_ind(Request, Xtl::Error, XtlKVList&);
private:
    XtlAddress _remoteNumber;
};


#endif /* _XTL_VOICECALL_H */
```

Code Example A-2 Listing of `voicecall.cc`

```
// Copyright 1993 by Sun Microsystems, Inc.
#pragma ident "@(#)voicecall.cc1.2894/02/22SMI"

//
//
// DirectAudioCall is an example of a higher level, easier to
// use XtlCall object. It simplifies the process of making
// and receiving a call, and configuring the call's data stream.
//

#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>

#include <xtl/xtl.h>
#include "voicecall.h"


//  The following code defines an XTL DirectAudioCall object's
//  response to being activated_ind(). The XTL platform calls
//  DirectAudioCall::activated_ind() when this call is activated.
//
//  Switching on the state of this call this function either
//  makes an outgoing call, answers an incoming call,
//  or correctly configures the data stream for the given state.
//
void
DirectAudioCall::activated_ind(XtlKVList&)
{
    XtlFormatempty_format;
```

Code Example A-2 Listing of `voicecall.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
    Exception excp;
    switch (call_state(excp).state()) {
    case IDLE:
    connect_req(
     "",
     _remoteNumber,
     empty_format);
    break;
    case INCOMING:
    answer_req();
    break;
    // unsupported by DirectAudioCall
    case DISCONNECTED:
    case INVALID:
    default:
    fprintf(stderr,
     "DirectAudioCall: given call in unsupported state\n");
    }

    if (call_state(excp).media_channel_available()) {
    send_desired_config();
    }
}


//  The DirectAudioCall event handler is a notification method.
//  It is called whenever the call's state changes.
//
void
DirectAudioCall::event_ind(CallEvent ev, Cause, XtlKVList&)
{
    // if data stream is available, then configure the data stream
    if (ev == CHANNEL_AVAILABLE_EVENT)
    send_desired_config();

    // ignore all other events
}


//  Send the desired data stream configuration based on the
//  call state.
//  The standard configuration is to connect to the providers
//  default microphone and speaker, but applications
```

Code Example A-2 Listing of `voicecall.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
//  can override this function and set up their own
//  configuration as well.
//
void
DirectAudioCall::send_desired_config()
{
    // ACTIVE_CALL indicates that the data stream is available.
    // Attempt to connect data stream to the providers
    // default microphone and speaker.
    //
    XtlKVList default_config;
    Exception excp;
    if (call_state(excp).media_channel_available()) {
    XtlStringinput;
    XtlStringoutput;
    XtlKVListdefaults;
    if (xtl_provider_info(call_state(excp).provider()->name(excp),
     defaults) < 0) {
        fprintf(stderr, "Provider Database not configured?\n");
        return;
    }
    if (!defaults.first("DEFAULT_MICROPHONE") ||
     !defaults.get(input)) {
        fprintf(stderr, "Default Input not found.\n");
        return;
    }
    if (!defaults.first("DEFAULT_SPEAKER") ||
     !defaults.get(output)) {
        fprintf(stderr, "Default Output not found.\n");
        return;
    }

    default_config.add(XtlConfigInputK, input());
    default_config.add(XtlConfigOutputK, output());

    // This configureDataStream method configures the data stream
    // using the key-value pairs contained in the default_config
    // argument.
    configuration_req(default_config);
    }
}

//  this returns the new valid configurations, else error() is called.
```

Code Example A-2 Listing of `voicecall.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
void
DirectAudioCall::configuration_ind(XtlKVList& new_config)
{
    fprintf(stderr, "New Data Steam Configuration:\n");
    new_config.print(STDERR_FILENO);
}


void
DirectAudioCall::extension_ind(
    const XtlString& message_name,
    XtlKVList& provider_extension)
{
    fprintf(stderr,
     "Unexpected Extension message: \"%s\"\n", message_name());
    provider_extension.print(STDERR_FILENO);
}

//  prints asynchronous errors to stderr.
void
DirectAudioCall::error_ind(Request req, Xtl::Error err, XtlKVList& )
{
    XtlStringrequest = string(req);
    XtlStringerror = Xtl::string(err);

    fprintf(stderr, "Request %s failed: %s\n", request(), error());
}
```

Answering Incoming Calls

Code Example A-3 shows the `incall.cc` program, which answers an incoming voice call. This program uses the `voicecall.h` header file described in “Handling Audio” on page 95.

Code Example A-3 Listing of `incall.cc`

```
// Copyright 1993 by Sun Microsystems, Inc.
#pragma ident "@(#)incall.cc1.2694/02/22SMI"


//
//  incall
//
//  Usage:incall [provider_name]
//
//  This is a short example which starts up a provider (either
//  specified on the command line, or the default provider),
//  listens for calls coming in through that provider, and
//  answers incoming calls.
//

#include <xtl/xtl.h>
#include <xtl/xtlprovider.h>
#include <xtl/xtlcall.h>
#include <xtl/xtlcallstate.h>
#include <Dispatch/sldispatcher.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

#include "voicecall.h"

//
// DirectAudioCall (defined in voicecall.h) is derived from the
// XtlCall class. We will keep one call for our provider.
//

DirectAudioCall*audiocall=0;

//
// MyProvider class declaration
//
```

Code Example A-3 Listing of `incall.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
//
// The MyProvider class is derived from the XtlProvider class (defined
// in xtlprovider.h) and provides implementations for methods to claim
// the call, watch its state progress, and become activated or
// deactivated. The MyProvider constructor invokes the XtlProvider
// constructor to get proper behavior.
//

class MyProvider : public XtlProvider {
public:
    MyProvider(Xtl::Exception& err, XtlString pname) : XtlProvider(err, pname)
{}
    virtual void activated_ind(XtlKVList&);
    virtual void deactivated_ind(XtlKVList&);
    virtual void offer_ind(XtlCallState& cs, XtlKVList&);
    virtual void call_event_ind(
     XtlCallState&, CallEvent, Cause, XtlKVList&);
};

//
//
// MyProvider class definition ////////////////////////////////
//
//

//
// The callOfferEvent method is invoked when a call becomes available.
// When this occurs, MyProvider::callOfferEvent destroys any previously
// existing call and accepts the new call in its place.
//
void
MyProvider::offer_ind(XtlCallState& call, XtlKVList&)
{
    // pick up any call offered to us
    if (audiocall)
    delete audiocall;
    Xtl::Exception e;
    audiocall = new DirectAudioCall(e, call);
}

//
// The activated method is invoked when an instance of the MyProvider
// class is successfully initialized. It prints out notice of what
```

Code Example A-3 Listing of `incall.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
// provider it has used, and then expresses interest in any incoming
// call.
//
void
MyProvider::activated_ind(XtlKVList&)
{
    // get provider name and print it out
    Exception excp;
    XtlStringprovider_name = name(excp);
    fprintf(stderr, "Using provider: %s\n", provider_name());

    // register for incoming call events
    enable_offer_event_req(B_TRUE);

    // listen for any call being disconnected
    listen_req(DISCONNECT_EVENT);
}

//
// The deactivated method is invoked when the provider goes away. It
// prints an appropriate notification message.
//
void
MyProvider::deactivated_ind(XtlKVList&)
{
    fprintf(stderr, "Provider died.\n");
    exit (1);
}

//
// The callevent method is invoked when the state of a call changes
// The provider receives notification identifing the call, and
// the event which identifies the state that changed.
//
void
MyProvider::call_event_ind(
 XtlCallState& ,
 CallEvent ev,
 Cause,
 XtlKVList&)
{
 // convert the event values into human readable string
 XtlStringevent = Xtl::string(ev);
```

Code Example A-3 Listing of `incall.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.

 fprintf(stderr, "Provider event = %s\n", event());
}

main(int argc, char* argv[])
{
    MyProvider::Exceptionerr;// To hold provider creation error
    MyProvider*Pv;
    dpDispatcher::instance(new dpSLDispatcher);
    dpDispatcher& d = dpDispatcher::instance();

    // Parse arguments
    if (argc > 2) {
    fprintf(stderr, "usage: %s [provider]", argv[0]);
    exit(1);
    }

    char*   pvname;
    if (argc == 2) {
    pvname = argv[1];// Use specified provider
    } else {
    pvname = NULL;// Use default provider
    }

    // Connect to provider
    Pv = new MyProvider(err, pvname);
    if (err != MyProvider::EXCEPTION_SUCCESS) {
    fprintf(stderr, "could not connect to provider\n");
    exit(1);
    }

    // enter the dispatch loop
    while(1)
     d.dispatch();
}
```

Using the Dispatcher and Notifier Interfaces

Code Example A-4 shows how to use the dispatcher.

Code Example A-4 Dispatcher Example.

```
#include "xv_dispatch.h"
extern "C" {
#include <xview/notify.h>
}


Notify_value input_wrapper(Notify_client, int fd) {
    dpSLDispatcher& d = Dispatcher::instance();

    if (d.setReady(fd, Dispatcher::ReadMask))
    d.dispatch();

    return NOTIFY_DONE;
}

Notify_value output_wrapper(Notify_client, int fd) {
    Dispatcher& d = Dispatcher::instance();

    if (d.setReady(fd, Dispatcher::WriteMask))
    d.dispatch();

    return NOTIFY_DONE;
}


void XVDispatcher::attach(int fd, Dispatcher::DispatcherMask mask, IOHandler*
handler) {
    Dispatcher::attach(fd, mask, handler);

    switch (mask) {
    case Dispatcher::ReadMask:
        notify_set_input_func((Notify_client) this,
                 (Notify_func) input_wrapper,
                 fd);
        break;
    case Dispatcher::WriteMask:
        notify_set_output_func((Notify_client) this,
```

```
                (Notify_func) output_wrapper,
                fd);
        break;
    }
}

void XVDispatcher::detach(int fd) {
    Dispatcher::detach(fd);

    notify_set_input_func((Notify_client) this, NOTIFY_FUNC_NULL, fd);
    notify_set_output_func((Notify_client) this, NOTIFY_FUNC_NULL, fd);
}
```

Creating an Answering Machine

Code Example A-6 (`machine.cc`) creates an answering machine client; it uses the `msgcall.h` header file shown in Code Example A-5.

Code Example A-5 Listing of `msgcall.h`

```
/* Copyright 1993 by Sun Microsystems, Inc. */
#pragma ident "@(#)msgcall.h1.2394/02/08SMI"

#ifndef _XTL_MSGCALL_H
#define _XTL_MSGCALL_H


#include <xtl/xtlcall.h>
#include <datapump/datapump.h>
#include <Dispatch/iohandler.h>
#include <datapump/dtmfdet.h>

class AudioWriter;
class DTMFHandler;

//  MsgCall is simple object for recording and playing Sun
//  audio files using a call's data stream.
//  Play and record can not be done simultaneously.
//
//  playDone()Detects the end of a playAnnc(). Relies on Sun
//      audio (4) driver interface. Only known to work on
//      xtlp_sun_5e5 provider.
```

Code Example A-5 Listing of `msgcall.h` *(Continued)*

```
/* Copyright 1993 by Sun Microsystems, Inc. */
//  playAnnc() Starts playing the announcement file.
//  recordMsg() Recording into the message file AFTER TRUNCATING IT.
//  stopPlayAnnc()Stops the play.
//  stopRecordMsg()Stops recording, closes file, and adds an audio header.
//
class MsgCall : public XtlCall {
public:
    MsgCall(
     Xtl::Exception e,
     XtlCallState& cs, // Claimable call
     char* announcement, // Full path to audio file to play
     char* message);// Full path to record file
    virtual ~MsgCall();
    virtual void configuration_ind(XtlKVList& newconfig);
    virtual void error_ind(Request req, Xtl::Error err, XtlKVList&);
    virtual void playDone()=0;
    virtual boolean_t playAnnc();
    virtual void stopPlayAnnc();
    virtual boolean_t recordMsg();
    virtual void stopRecordMsg();
    XtlString messageFile();
    XtlString announcementFile();
private:
    void start_play();
    void start_record();
    int _audioFd;
    int _anncFd;
    int _msgFd;
    XtlString_anncAudioFile;
    XtlString_msgAudioFile;

    DataPump*audioPump;
    AudioWriter*player;
    CopyReader*recorder;
    boolean_trecord_requested;
    boolean_tplay_requested;
    DTMFHandler*dtmf;
};

inline XtlString MsgCall::messageFile() { return _msgAudioFile; }
inline XtlString MsgCall::announcementFile() { return _anncAudioFile; }

// DTMFHandler is used by MsgCall to detect dtmf.
```

Code Example A-5 Listing of `msgcall.h` *(Continued)*

```
/* Copyright 1993 by Sun Microsystems, Inc. */
// If just the dtmf tone or sting detected.
//
class DTMFHandler : public DTMFReader {
public:
    DTMFHandler(DataPump& audioPump) : DTMFReader(audioPump) {}
    virtual void detectedToneUp(char c);
    virtual void detectedToneDown(char c);
    virtual void detectedToneString(XtlString& tones);
};

//
// AudioWriter is used to play a Sun audio file to a Sun audio device.
//
//  NOTE:AudioWriter is designed to work on the Sun /dev/audio
//  (audio (4)) interface. Some providers configurations
//  may not support the ioctls used by AudioWriter. This
//  example is only known to work on the xtlp_sun_5e5
//  configured as INPUT = STREAM, and OUTPUT = STREAM.
//
//  The AudioWriter client_data is a pointer to a composite object that
//  is using the AudioWriter. It is used in donePlaying() to notify the
//  composite object that the play has finished. By default the
//  AudioWriter assumes the composite object is MsgCall, but AudioWriter
//  could be subclassed, and donePlaying could be overridden to call
//  a different composite object.
//
class AudioWriter : public CopyWriter, public dpIOHandler {
public:
    AudioWriter(DataPump& audio, int file_fd, void* client_data = NULL,
     int bufsize = 1024);
    virtual ~AudioWriter();
    virtual void donePlaying();
    virtual void* clientData();
protected:
    boolean_t playedZeroLengthBuffer(int fd);
    virtual void timerExpired(long sec, long usec);
private:
    int audio_device;
    void*client_data;
};


#endif /* _XTL_MSGCALL_H */
```

Code Example A-6 Listing of `machine.cc`

```
// Copyright 1993 by Sun Microsystems, Inc.
#pragma ident "@(#)machine.cc1.2094/02/22SMI"



#include <xtl/xtl.h>
#include <xtl/xtlprovider.h>
#include <xtl/xtlcall.h>
#include <xtl/xtlcallstate.h>
#include <Dispatch/sldispatcher.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

#include "msgcall.h"

//
//  usage: machine [provider]
//
//  Machine immediately answers all incoming calls, plays a
//  greeting file from the /usr/demo/SOUND directory, and
//  then records a mesage.
//

MsgCall*msgcall; // We only handle one global call at a time



// MyCall is a class which will claim and answer a call,
// play a greeting, then take a message.

class MyCall : public MsgCall {
public:
    MyCall(
     Xtl::Exception& e,
     XtlCallState& cs,
     char* announcement,
     char* message) :
     MsgCall(e, cs, announcement, message) {}
    virtual void playDone();
    virtual void activated_ind(XtlKVList&);
```

Code Example A-6 Listing of `machine.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
    virtual void event_ind(CallEvent ev, Cause, XtlKVList&);
};


// This is a notification which has been overridden.
// The call object now has ownership of the call. The "activated_ind()"
// notification only gets called once during the lifetime of a call object.

void
MyCall::activated_ind(XtlKVList&)
{
 fprintf(stderr,
    "MyCall: activated \n");

    Exception excp;
    switch (call_state(excp).state()) {
    case INCOMING:
    // The call is incoming and has not been
    // answered by another call object.
    answer_req();
    break;

    // unsupported by MsgCall
    case UNKNOWN:
    case IDLE:
    case DISCONNECTED:
    case INVALID:
    default:
    fprintf(stderr,
     "MyCall: given call in unsupported state\n");
    }

    // The phone call may have been previously answered by another
    // call object but then had its ownership offered. This call
    // object now has claimed ownership of the call and will play
    // the greeting. It is not necessary to call "answer_req()" the
    // call because it is not incoming or in one of the inactive states.
    // Just check if the data channel is available, if so start play
    if (call_state(excp).media_channel_available())
    playAnnc();
}
```

Code Example A-6 Listing of `machine.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
// This notification method gets called when the greeting file has finished
// playing. It is now time to record the caller's message.
void
MyCall::playDone()
{
    recordMsg();
}

// The "event()" notification function gets called when there is a
// change in the state of a call. This function will be called after
// the "activated_ind()" function has been called. The event notification
// is usually called several times during the course of a call. The
// sequence of events for a given call is considered to be
// indeterminate.

void
MyCall::event_ind(CallEvent ev, Cause, XtlKVList&)
{
    switch(ev) {
    case CHANNEL_AVAILABLE_EVENT:
     // The call object may now attempt to acquire a
    // data stream fd and play a greeting.
    playAnnc();
    break;
    // ignored by MsgCall
    case CONNECT_EVENT:
    case ALERTING_EVENT:
    case PROCEEDING_EVENT:
    break;
    case DISCONNECT_EVENT:
    // The call has been disconnected and it is time
    // to stop recording the call's data stream and
    // delete this object.
    stopRecordMsg();

    // this code only manages one call at a time, delete
    // only call and reset global pointer
    delete this;
    msgcall = NULL; // Set the global msgcall pointer to null.
    break;

    // unsupported by MyCall
    case CHANNEL_UNAVAILABLE_EVENT:
```

Code Example A-6 Listing of `machine.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
    default:
    fprintf(stderr,
     "MyCall: ignoring %s Event\n", Xtl::string(ev)());
    }
}


////////////////////////////////////////////////////////////////////////

// MyProvider is a subclass of XtlProvider designed to wait for an offered
// call and then claim it and take a message.

class MyProvider : public XtlProvider {
public:
    MyProvider(Xtl::Exception& err, XtlString pname) : XtlProvider(err, pname)
{}
    virtual void activated_ind(XtlKVList&);
    virtual void deactivated_ind(XtlKVList&);
    virtual void offer_ind(XtlCallState& cs, XtlKVList&);
    virtual void call_event_ind(
     XtlCallState&, CallEvent, Cause, XtlKVList&);
};

// A call has been offered for ownership and the MyProvider object will
// attempt to claim it by creating a new call object using the call state
// of the offered call.
void
MyProvider::offer_ind(XtlCallState& call, XtlKVList&)
{
    fprintf(stderr, "got callOfferEvent\n");

    // pick up incoming call
    if (call.state() == INCOMING) {
    fprintf(stderr, "incoming call....\n");
    // Claim the call and tell it to use train.au for the greeting
    // message and msg.au for the message file.
    Xtl::Exception e;
    msgcall = new MyCall(e, call,
     "/usr/demo/SOUND/sounds/train.au", "/tmp/msg.au");
    }
}

// This notification is called when the provider object has finished
```

Code Example A-6 Listing of `machine.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
// its internal setup and is ready for commands or further notifications.
void
MyProvider::activated_ind(XtlKVList&)
{
    Exception excp;
    fprintf(stderr, "Using provider: %s\n", (name(excp))());

    // register for offer events
    enable_offer_event_req(B_TRUE);
}

// This notification is called when the provider process associated with this
// provider object dies.
void
MyProvider::deactivated_ind(XtlKVList&)
{
    fprintf(stderr, "Provider died.\n");
    exit(1);
}

// This provider object could be instructed to listen for call events
// using the listen_req() command. Such call events would be observed
// using this notification.
void
MyProvider::call_event_ind(XtlCallState& , CallEvent ev, Cause, XtlKVList&)
{
    // Not currently listening for any provider events
    //
    fprintf(stderr, "MyProvider:: Ignoring %s provider event\n",
     Xtl::string(ev)());
}

main(int argc, char* argv[])
{
    MyProvider::Exceptionerr;
    MyProvider*Pv;
    dpDispatcher::instance(new dpSLDispatcher);
    dpDispatcher& d = dpDispatcher::instance();

    // Parse arguments
    if (argc > 2) {
    fprintf(stderr, "usage: %s [provider]", argv[0]);
    exit(1);
```

Code Example A-6 Listing of `machine.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
    }

    // Assume that the second argument (if it exists) to the program
 // is the provider name.
    char*   pvname;
    if (argc == 2) {
    pvname = argv[1];
    } else {
    pvname = NULL;
    }

    // Connect to the provider specified by pvname. If pvname is NULL
    // then XTL will attempt to start up the default system provider.
    Pv = new MyProvider(err, pvname);
    if (err != MyProvider::EXCEPTION_SUCCESS) {
    fprintf(stderr, "could not connect to provider\n");
    exit(1);
    }

    // Sit in the dispatcher loop
    while(B_TRUE)
    d.dispatch();
}
```

To test the answering machine from Code Example A-6, use the `msgcall.cc` program shown in Code Example A-7, which makes calls to the answering machine; refer to Code Example A-5 on page 106 for the header file for `msgcall.cc`.

Code Example A-7 Listing of `msgcall.cc`

```
// Copyright 1993 by Sun Microsystems, Inc.
#pragma ident "@(#)msgcall.cc1.3094/02/22SMI"




#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <stdio.h>
#include <strings.h>
#include <stropts.h>

#include <multimedia/libaudio.h>

#include <xtl/xtl.h>

#include "msgcall.h"

//
// AudioWriter is used to play a Sun audio file to a Sun audio device.
//
//  audio = DataPump attached to a Sun audio STREAMS device
//  audio_file = file descriptor to a Sun audio file.
//  bufsize = size of buffer to use when copying from file to device
//
//  NOTE:AudioWriter is designed to work on the Sun /dev/audio
//  (audio (4)) interface. Some providers configurations
//  may not support the ioctls used by AudioWriter. This
//  example is only known to work on the xtlp_sun_5e5
//  configured as INPUT = STREAM, and OUTPUT = STREAM.
//
//  The AudioWriter client_data is a pointer to a composite object that
//  is using the AudioWriter. It is used in donePlaying() to notify the
//  composite object that the play has finished. By default the
//  AudioWriter assumes the composite object is MsgCall, but AudioWriter
//  could be subclassed, and donePlaying could be overridden to call
//  a different composite object.
//
```

Code Example A-7 Listing of `msgcall.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
AudioWriter::AudioWriter(
    DataPump& audio,
    int audio_file,
    void* data,
    int bufsize)
    : CopyWriter(audio, audio_file, bufsize),
     audio_device(audio.fd()),
     client_data(data)// a MsgCall
{
    dpDispatcher& d = dpDispatcher::instance();

    // clear eof flag of audio device, so we can detect end of play
    audio_info_t audioinfo;
    AUDIO_INITINFO(&audioinfo);// init info struct
    audioinfo.play.eof = 0;// clear eof flag
    audio_setinfo(audio.fd(), &audioinfo);

    d.startTimer(0, 100, this);
}

AudioWriter::~AudioWriter()
{
    dpDispatcher& d = dpDispatcher::instance();

    // Stop the timer
    d.stopTimer(this);
}


void* AudioWriter::clientData() { return client_data; }

//
// The outgoing message is done playing.
//
void
AudioWriter::donePlaying()
{
    // Specific to using MsgCall, subclass and override this function
    // to call a different composite object
    MsgCall* msgcall = (MsgCall*) clientData();
    msgcall->stopPlayAnnc();
    msgcall->playDone();
}
```

Code Example A-7 Listing of `msgcall.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.

//
// AudioWriter determines that the outgoing message has been completely
// played if the audioinfo.play.eof bit associated with the fd
// is 1.
//
boolean_t
AudioWriter::playedZeroLengthBuffer(int fd)
{
 audio_info_t audioinfo;
 int err;


 err = audio_getinfo(fd, &audioinfo);
 if ((err == 0) && (audioinfo.play.eof >= 1)) {
 /* done playing */
    return(B_TRUE);
 }

 return(B_FALSE);
}

//
// Interrupt handler for the timer set in the AudioWriter constructor.
// Check to see if the outgoing message is done playing. If it hasn't
// reset the timer.
//
void
AudioWriter::timerExpired(long, long)
{
    if (playedZeroLengthBuffer(audio_device) == B_TRUE)
    donePlaying();
    else {
    // reset timer
    dpDispatcher& d = dpDispatcher::instance();
    d.startTimer(0, 100, this);
    }
}


// MsgCall will play a message (from a file referenced by the announce
// parameter. or record a message (to a file referenced by the message
// parameter) over a call's data channel. It plays a message when
```

Code Example A-7 Listing of `msgcall.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
// MsgCall::playAnnc is called and records a message when MsgCall::recordMsg
//is called.
//
MsgCall::MsgCall(
    Xtl::Exception e,
    XtlCallState&cs,
    char* announcement,
    char*message)
    : XtlCall(e, cs), _audioFd(-1), _anncFd(0), _msgFd(0), audioPump(0),
    player(0), recorder(0), record_requested(B_FALSE),
    play_requested(B_FALSE), dtmf(0), _anncAudioFile(announcement),
    _msgAudioFile(message)
{
}

MsgCall::~MsgCall()
{
    fprintf(stderr, "MsgCall: cleaning up\n");

    // stop any records or plays that may be going on
    stopPlayAnnc();
    stopRecordMsg();

    close(_audioFd);
    close(_anncFd);
    close(_msgFd);
}

void
MsgCall::error_ind(Request req, Xtl::Error err, XtlKVList&)
{
    XtlStringrequest = string(req);
    XtlStringerror = Xtl::string(err);

    fprintf(stderr, "Request %s failed: %s\n", request(), error());
}

//
// find_kv_string_pair is an auxiliary function used to search an XtlKVList
// for a kv pair whose key is <key> and whose value (an XtlKVString)
// is <value>.
//
boolean_t
```

Code Example A-7 Listing of `msgcall.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
find_kv_string_pair(
    XtlKVList kv_list,
    const XtlString &key,
    const XtlString &value)
{
    XtlStringthis_value;

    kv_list.reset();
    while (kv_list.next(key) == B_TRUE) {
    if (kv_list.get(this_value) == B_FALSE) {
        continue;
    }
    if (this_value == value) {
        return(B_TRUE);
    }
    }
    return(B_FALSE);
}



//
// When the call's data stream is reconfigured to include INPUT=STREAM and
// OUTPUT=STREAM then play the greeting message or record an
// imcoming message depending on whether record_requested or play_requested
// is set (set during playAnnc and recordMsg, respectively).
//
// If the data stream configuration is not INPUT=STREAM and OUTPUT=STREAM,
// there is trouble. The record or play machinery should be cleaned up.
// But for simplicity sake, we just exit instead.
//
void
MsgCall::configuration_ind(XtlKVList& config)
{
    // ignore null configuration event
    if (!config.first())
    return;

    if ((find_kv_string_pair(config, XtlConfigInputK, XtlConfigStreamC) !=
B_TRUE) &&
     (find_kv_string_pair(config, XtlConfigOutputK, XtlConfigStreamC) !=
B_TRUE)) {
    fprintf(stderr, "Warning: Unsupported configuration:\n");
```

Code Example A-7 Listing of `msgcall.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
    config.print(STDERR_FILENO);
    fprintf(stderr, "Exiting\n");
    exit(1);
    }

    // pull input and output fds out of congfiguration, and save them
    config.first("INPUT_STREAM_FD");
    u_longinput_fd;
    config.get(input_fd);
    config.first("OUTPUT_STREAM_FD");
    u_longoutput_fd;
    config.get(output_fd);

    // MsgCall code assumes we have a bidirectional stream. If
    // we don't get a bidirectional stream back, exit
    //
    if (output_fd != input_fd) {
    fprintf(stderr,
     "detected unidirectional stream, exiting.\n");
    exit(1);
    }

    // save valid audio fd
    _audioFd = int(output_fd);

    // Sanitiy test
    if ((record_requested == B_TRUE) && (play_requested == B_TRUE)) {
    fprintf(stderr,
     "internal error: recording and playing simultaneously\n");
    exit(1);
    }

    // proceed with existing request
    if (record_requested) {
    start_record();
    } else if (play_requested){
    start_play();
    }
}



//
```

Code Example A-7 Listing of `msgcall.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
// Start recording (or prepare the data configuration to record) the
// callers message.
//
// RETURNs B_FALSE if object is already busy, B_TRUE on success.
//
boolean_t
MsgCall::recordMsg()
{
    XtlKVListread_write_stream_config;

    // If we are already doing something return an error
    Exception excp;
    if ((play_requested) || (record_requested) ||
     !call_state(excp).media_channel_available()) {
    return B_FALSE;
    }

    record_requested = B_TRUE;

    // _audioFd is a file descritor to the call's data channel.
    // If it is not a valid fd, configure the data stream
    // to INPUT=STREAM and OUTPUT=STREAM mode. configuration_ind
    // will set _audioFd approriately.
    // Otherwise start recording.
    if (_audioFd < 0) {
    read_write_stream_config.add(XtlConfigInputK, XtlConfigStreamC);
    read_write_stream_config.add(XtlConfigOutputK, XtlConfigStreamC);

    configuration_req(read_write_stream_config);
    } else {
    start_record();
    }

    return B_TRUE;
}

//
// Start playing (or prepare the data configuration to play) the outgoing
// (greeting) message
//
// RETURNs B_FALSE if object is already busy, B_TRUE on success.
//
boolean_t
```

Code Example A-7 Listing of `msgcall.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
MsgCall::playAnnc()
{
    XtlKVListread_write_stream_config;

    // If we are already doing something return an error
    Exception excp;
    if ((play_requested) || (record_requested) ||
     !call_state(excp).media_channel_available()) {
    return B_FALSE;
    }

    play_requested = B_TRUE;

    // _audioFd is a file descriptor to the call's data channel.
    // If it is not a valid fd, configure the data stream
    // to INPUT=STREAM and OUTPUT=STREAM mode. configuration_ind
    // will set _audioFd appropriately.
    // Otherwise start playing.
    if (_audioFd < 0) {
    read_write_stream_config.add(XtlConfigInputK, XtlConfigStreamC);
    read_write_stream_config.add(XtlConfigOutputK, XtlConfigStreamC);

    configuration_req(read_write_stream_config);
    } else {
    start_play();
    }

    return B_TRUE;
}

//
// Start recording an incoming message to _msgAudioFile (as initialized
// in MsgCall constructor).
//
void
MsgCall::start_record()
{
    if ((_msgFd =
     open(_msgAudioFile(), O_WRONLY|O_TRUNC|O_CREAT, 0664)) < 0) {
    perror("MsgCall:: could not open message file");
    disconnect_req();
    }
```

Code Example A-7 Listing of `msgcall.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
    fprintf(stderr, "recording message...\n");

    // Use a DataPump and CopyReader to do the work
    audioPump = new DataPump(_audioFd);
    ioctl(_audioFd, I_FLUSH, FLUSHRW);
    recorder = new CopyReader(*audioPump, _msgFd);

    // add software dtmf filter
    dtmf = new DTMFHandler(*audioPump);
}

//
// Start playing the greeting message _anncAudioFile
//
void
MsgCall::start_play()
{
    // open outgoing announcement file
    if ((_anncFd <= 0)
     && (_anncFd = open(_anncAudioFile(), O_RDONLY)) < 0) {
    perror("could not open announcement file");
    disconnect_req();
    }

    // NOTE:
    //We assume that the file is a Sun audio format file, and
    //the device is a Sun audio device interface. This is
    //not very portable. For example, it would break if
    //the device was alaw and the file was ulaw. Ideally,
    //we would get the format of the data, and verify the
    //device can handle the format. Then if there were
    //an incompatiblity we could either convert the file format
    //to something the device could handle or change
    //the device's format mode.
    //

    // seek past audio file header
    Audio_hdraudio_header;
    if (AUDIO_SUCCESS !=
     audio_read_filehdr(_anncFd, &audio_header, NULL, 0)) {
    fprintf(stderr, "WARNING: not a Sun audio file\n");
    }
```

Code Example A-7 Listing of `msgcall.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
    fprintf(stderr, "playing message...\n");

    // create Xtl data stream pump, and flush Xtl data stream
    audioPump = new DataPump(_audioFd);
    ioctl(_audioFd, I_FLUSH, FLUSHRW);
    player = new AudioWriter(*audioPump, _anncFd, this);

    // add software dtmf filter
    dtmf = new DTMFHandler(*audioPump);
}

//
// When done recording the incoming message, add an audio header
// and clean up the recording machinery.
//
void
MsgCall::stopRecordMsg()
{
    // clean up must be done in this order
    close(_msgFd);
    _msgFd = -1;

    if (recorder) {
    // add header to raw audio file
    char   cmd[256];
    sprintf(cmd,
     "audioconvert -f voice -p -i rate=8k,channels=mono,encoding=ulaw %s",
     _msgAudioFile());
    if (system(cmd) < 0) {
        perror("could not convert message file");
    }
    }

    delete dtmf;
    dtmf = NULL;

    delete recorder;
    recorder = NULL;

    delete audioPump;
    audioPump = NULL;

    record_requested = B_FALSE;
```

Code Example A-7 Listing of `msgcall.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
}

//
// Clean up machinery used to play the greeting
//
void
MsgCall::stopPlayAnnc()
{
    // clean up must be done in this order
    close(_anncFd);
    _anncFd = -1;

    delete dtmf;
    dtmf = NULL;

    delete player;
    player = NULL;

    delete audioPump;
    audioPump = NULL;

    play_requested = B_FALSE;
}

//
// Detect when DTMF goes off (is released)
//
void
DTMFHandler::detectedToneUp(char c)
{
    fprintf(stderr, "detected up %c\n", c);
}

//
// Detect a string of DTMF tones (called when the "#" is detected)
//
void
DTMFHandler::detectedToneString(XtlString& tones)
{
    fprintf(stderr, "detected tone string %s\n", tones());
}

//
```

Code Example A-7 Listing of `msgcall.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
// Detect when DTMF goes on (is pressed)
//
void
DTMFHandler::detectedToneDown(char c)
{
    fprintf(stderr, "detected down %c ...", c);
}
```

Monitoring Calls

Code Example A-8 uses the `XtlMonitor` class to monitor calls.

Code Example A-8 Listing of `monitorcalls.cc`

```
// Copyright 1993 by Sun Microsystems, Inc.
#pragma ident "@(#)monitorcalls.cc1.3294/02/22SMI"


#include <xtl/xtl.h>
#include <xtl/xtlprovider.h>
#include <xtl/xtlmonitor.h>
#include <xtl/xtlcallstate.h>
#include <Dispatch/sldispatcher.h>
#include <stdio.h>
#include <stdlib.h>

//
// This program monitors all calls for a specified provider.
// It registers interest in all existing calls at start up, and
// then registers all future incoming and outgoing calls.
// If no provider is given the default provider is used.
//
// usage: monitorcalls [<provider-name>]
//

class myProvider : public XtlProvider {
public:
    myProvider(Xtl::Exception& err, XtlString pname) : XtlProvider(err, pname)
{}
    virtual void activated_ind(XtlKVList&);
```

Code Example A-8 Listing of `monitorcalls.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
    virtual void deactivated_ind(XtlKVList&);
    virtual void offer_ind(XtlCallState&, XtlKVList&);
    virtual void call_event_ind(
     XtlCallState&, CallEvent, Cause, XtlKVList&);
    virtual void list_calls_ind(
     XtlCallState *const * list,
     u_int length);
};

class myMonitor : public XtlMonitor {
public:
    myMonitor(Xtl::Exception& e, XtlCallState& cs)
     : XtlMonitor(e, cs) {}
    virtual void activated_ind(XtlKVList&);
    virtual void deactivated_ind(XtlKVList&);
    virtual void event_ind(CallEvent, Cause, XtlKVList&);
};

//
// When the provider object is activated, request a list of all the
// existing calls, and register interest in the creation of any new calls.
//
void
myProvider::activated_ind(XtlKVList&)
{
    Exception excp;
    fprintf(stdout, "Using provider: %s\n", (name(excp))());

    // get list of existing calls
    list_calls_req();

    // register for calls available for ownership
    enable_offer_event_req(B_TRUE);

    // register for new call creation
    listen_req(Xtl::CREATE_CALL_EVENT);

    // register for these events on any call
    //listen_req(Xtl::PROCEEDING_EVENT);
    //listen_req(Xtl::ALERTING_EVENT);
    //listen_req(Xtl::CONNECT_EVENT);
    //listen_req(Xtl::FAILURE_EVENT);
    //listen_req(Xtl::DISCONNECT_EVENT);
```

Code Example A-8 Listing of `monitorcalls.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.

    //listen_req(Xtl::CHANGE_OWNER_EVENT);
    //listen_req(Xtl::TRANSFER_EVENT);
    //listen_req(Xtl::REDIRECT_EVENT);
    //listen_req(Xtl::CONFERENCE_EVENT);
    //listen_req(Xtl::DROP_EVENT);
    //listen_req(Xtl::INFO_EVENT);

    //listen_req(Xtl::INVALIDATE_CALL_EVENT);
}

// Exit if the provider dies.
void
myProvider::deactivated_ind(XtlKVList&)
{
    fprintf(stderr, "Provider died.\n");
    exit(1);
}

// All incoming calls will be offered to the provider call for ownership.
// If the state of the call is incoming, it is a new call, so create
// another monitor. Any other state means we are already monitoring this
// call so ignore the event.
// NOTE: monitor object is deleted when its call is destroyed
//
void
myProvider::offer_ind(XtlCallState& cs, XtlKVList&)
{
    fprintf(stdout, "Offered Call Event,\tstate = %s\n",
     (Xtl::string(cs.state()))());
    if (cs.state() == INCOMING) {
    myMonitor*monitor;
    Xtl::Exception e;
    monitor = new myMonitor(e, cs);
    fprintf(stdout,
     "\nCallReference = \"%s\"\twas created (INCOMING)\n\n",
     (cs.call_reference())());
    }
}

// All outgoing calls will start with a created call event.
// If we detect a created call event, it is a new call, so create
// another monitor. Any other event means we are already monitoring this
```

Code Example A-8 Listing of `monitorcalls.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
// call so ignore the event.
// NOTE: monitor object is deleted when its call is destroyed
//
void
myProvider::call_event_ind(
 XtlCallState& cs,
 CallEvent cev,
 Cause,
 XtlKVList&)
{
    // Monitor all events on *this* call
    if (cev == CREATE_CALL_EVENT) {
    myMonitor*monitor;
    Xtl::Exception e;
    monitor = new myMonitor(e, cs);
    }

    // Print provider event for any call
    fprintf(stderr, "Provider Caught: %s,\ton CallReference: \"%s\"\n",
     Xtl::string(cev)(), cs.call_reference()());
}

// At provider start up we requested a list of all existing calls.
// This is the reply. Create a monitor for each existing call.
//
void
myProvider::list_calls_ind(
    XtlCallState *const * list,
    u_int length)
{
    myMonitor*monitor;

    // if no calls exist , return
    if (length == 0) {
    fprintf(stdout, "Waiting for new calls...\n");
    return;
    }

    fprintf(stdout, "Creating Monitors for these existing calls:\n");

    for (int i = 0; i < length; i++) {
    fprintf(stdout, "CallReference = \"%s\"\tstate = %s%s\n",
     list[i]->call_reference()(),
```

Code Example A-8 Listing of `monitorcalls.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
     Xtl::string(list[i]->state())(),
     list[i]->held() ? ", HELD" : "");

    // create a monitor for this call
    Xtl::Exception e;
    monitor = new myMonitor(e, *list[i]);
    }
    fprintf(stdout, "Waiting for new calls...\n");
}

void
myMonitor::activated_ind(XtlKVList&)
{
    Exception excp;
    XtlCallState&cs = call_state(excp);
    if (cs.state() == IDLE) {
    fprintf(stdout, "\nCallReference = \"%s\"\twas created\n\n",
     (cs.call_reference())());
    }
}

// If the call this object is monitoring is destroyed, or the provider dies,
// delete this object.
//
void
myMonitor::deactivated_ind(XtlKVList&)
{
    Exception excp;
    XtlCallState&cs = call_state(excp);
    fprintf(stdout, "\nCallReference = \"%s\"\twas destroyed\n\n",
     (cs.call_reference())());

    // self-deleting object
    delete this;
}

// This function gets notified of every event on its call.
// For each event, print the global call reference and the event.
//
void
myMonitor::event_ind(CallEvent ev, Cause, XtlKVList&)
{
    Exception excp;
```

Code Example A-8 Listing of `monitorcalls.cc` *(Continued)*

```
// Copyright 1993 by Sun Microsystems, Inc.
    XtlCallState&cs = call_state(excp);

    fprintf(stdout, "CallReference = \"%s\"\tgot event = %s\n",
     (cs.call_reference())(), (Xtl::string(ev))());
}


// Main body.
// Parse command line arguments, and create a provider.
// Then wait for events.
//
void
main(int argc, char* argv[])
{
    Xtl::Exceptionerr;
    myProvider*Pv;
    dpDispatcher::instance(new dpSLDispatcher);
    dpDispatcher& d = dpDispatcher::instance();
    char*   pvname;

    // Parse arguments
    if (argc > 2) {
    fprintf(stderr, "usage: %s [provider]", argv[0]);
    exit(1);
    }

    if (argc == 2) {
    pvname = argv[1];
    } else {
    pvname = NULL;
    }

    // Connect to provider
    Pv = new myProvider(err, pvname);
    if (err != myProvider::EXCEPTION_SUCCESS) {
    fprintf(stderr, "could not connect to provider\n");
    exit(1);
    }

    while(1)
    d.dispatch();
}
```

A

Index

D

E

F

G

N

O

P

Q

R

S

T

U

V

W

X

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 Application Programmer's Guide*, part number *801-7046-11*

■ 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 Application 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