Solaris Teleservices 1.0 API Programming Guide

EDR1

draft

Preliminary — 6/7/93

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

© 1993 Sun Microsystems, Inc.
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

THIS IS A PRELIMINARY SPECIFICATION SUBJECT TO CHANGE WITHOUT NOTICE. SunSoft, Inc. RESERVES THE RIGHT TO MODIFY THE SPECIFICATION AT ANY TIME WITHOUT PRIOR NOTFICATION.

All rights reserved. This product and related documentation are protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX® and Berkeley 4.3 BSD systems, licensed from UNIX System Laboratories, Inc. and the University of California, respectively. Third-party font software in this product is protected by copyright and licensed from Sun's Font Suppliers.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227-7013 (c)(1)(ii) and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS
Sun, Sun Microsystems, the Sun logo, SMCC, the SMCC logo, SunSoft, the SunSoft logo, Solaris, SunOS, OpenWindows, DeskSet, ONC, and NFS are trademarks or registered trademarks of Sun Microsystems, Inc. UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc. [ATTRIBUTION OF OTHER THIRD PARTY TRADEMARKS MENTIONED SIGNIFICANTLY THROUGHOUT PRODUCT OR DOCUMENTATION]. All other product names mentioned herein are the trademarks of their respective owners.

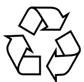
All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCworks, and SPARCompiler are licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK® and Sun™ Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

X Window System is a trademark and product of the Massachusetts Institute of Technology.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.



Please
Recycle

Contents

1. Introduction to the Solaris Teleservices API 1

The Solaris Teleservices API 2

XTEL API Characteristics 2

XTEL Events 3

Event Registration 3

XTEL Object Methods 3

Command Methods 3

Notification Methods. 3

Overview of XTEL Classes 4

2. Getting Started With XTEL Programming 7

Compiling XTEL Applications 7

Creating A Makefile. 7

How the XTEL Programming Interface Works 9

Programming Tips 10

Example Program outcall.cc 10

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

3. XTEL Classes **15**

Building Blocks 15

Using XtelCallReference Values 16

Events 16

XTEL Object Relationships 19

Object Activation and Deactivation. 19

XTEL Classes 19

Using XtelByteArray Objects 20

Using XtelString Objects 20

Using XtelKVList Objects 20

Using Dispatcher Objects 24

Using IOHandler Objects 25

Using XtelCallState Objects 26

Using XtelProvider Objects 27

Using XtelCall Objects. 30

Using XtelMonitor Objects 34

4. Creating XTEL Applications **37**

Creating a Basic XTEL Application 37

Header Files 38

Subclassing XTEL Classes. 39

Program Example outcall.cc 39

Handling Audio. 46

Answering Incoming Calls 50

Using the Dispatcher and Notifier Interfaces. 53

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Creating an Answering Machine 55

Monitoring Calls 62

5. Querying the XTEL Configuration Database 67

Overview. 67

Using the Database Query Functions 68

6. Using Data Streams 69

Manipulating Data Streams 69

Configuring Data Streams 70

Using Provider Specific Audio 71

Using System Default Audio 71

Using STREAMs 71

`Using` File Descriptors 72

Using Devices 72

Using Filters 72

Configuration Examples 72

Using Voice Services 74

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Tables

Table 3-1 CallState Events (from xtel.h) 17

Table 3-2 AdvisoryState Events (from xtel.h). 17

Table 3-3 CallEvent Events (from xtel.h). 18

Table 3-4 XtelKVList Class Command Methods 22

Table 3-5 Dispatcher Object functions. 24

Table 3-6 IOHandler Object functions. 26

Table 3-7 XtelCallState Class Command Methods 26

Table 3-8 XtelCallState Global Command Method (from xtelcallstate.h) 27

Table 3-9 XtelProvider Class Command Methods (from xtelprovider.h) 28

Table 3-10 XtelProvider Class Notification Methods (from xtelprovider.h) 28

Table 3-11 XtelProvider Request and Error Values (from xtelprovider.h) 30

Table 3-12 XtelCall Class Command Methods (from xtelcall.h) 31

Table 3-13 XtelCall Class Notification Methods (from xtelcall.h) 32

Table 3-14 XtelCall Request and Error Values (from xtelcall.h) 33

Table 3-15 XtelMonitor Command Method 34

Table 3-16 XtelMonitor Object State Methods 34

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Table 3-17 XtelMonitor Object Notification Methods 34
Table 3-18 Request, Error, and Event Parameter values: 35
Table 4-1 XTEL Header Files. 38
Table 5-1 Database Query Functions (from xtedb.h). 68
Table 6-1 Audio Inputs and Outputs for Key-Value Pairs 70
Table 6-2 Common Data Stream Configurations 73

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Figures

Figure 3-1 XtelKVList Structure 21
Figure 6-1 Data Stream Inputs and Outputs 70

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Preface

The *Solaris Teleservices 1.0 API Programming Guide* provides information on programming with the Solaris Teleservices Application Programmer's Interface (API), from here on referred to as XTEL.

Audience

This manual is for C++ programmers who are developing XTEL applications. You should have a good understanding of UNIX and the C++ programming language. To write window applications using XTEL, you should also be familiar with the OpenWindows™ windowing environment.

This manual provides example programs that help you to begin writing XTEL programs by illustrating concepts explained in the text.

Purpose of This Manual

This manual explains how to use XTEL to write applications that:

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

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Structure of This Manual

The chapters in this manual are organized as follows:

Chapter 1, "Introduction to the Solaris Teleservices API," introduces the XTEL API and XTEL objects.

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

Chapter 3, "XTEL Classes," explains the Solaris Teleservices programming model. It defines object-oriented programming and describes the provider and call objects.

Chapter 4, "Creating XTEL Applications," explains how to make and receive calls, then hold, drop, conference, forward, and transfer the call.

Chapter 5, "Querying the XTEL Configuration Database," defines the provider configuration information in the Solaris Teleservices database, and explains how to query the database.

Chapter 6, "Using Data Streams," explains, then describes how to access and use the data channel.

Appendix A, "Error Codes," shows the possible error codes returned by XTEL library calls.

Related Manuals

- *Solaris Teleservices 1.0 Architectural Overview*
- *Solaris Teleservices 1.0 System Administrator's Guide*
- *Solaris Teleservices 1.0 Installation Notes*

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

What Typographic Changes and Symbols Mean

The following table 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#`

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Introduction to the Solaris Teleservices API

The Solaris Teleservices application programming interface (API), called XTEL, 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, and redirect calls.

The XTEL API is a component of the larger Solaris Teleservices platform, which consists of the API, providers, and configuration databases. The platform is designed to operate using any switch or telephone technology. This platform independence is facilitated by *providers*, which hide the details of the underlying transmission technology.

Through XTEL, 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.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

The Solaris Teleservices API

The XTEL 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 XTEL API also uses an event-driven paradigm where your application is notified of asynchronous events generated by the Teleservices platform.

XTEL uses objects to represent the fundamental elements of a basic telephone platform. These elements are a phone device and the calls made using that device. An XtelProvider object represents a valid initialized phone device ready to make and receive calls. It hides specific device-dependent code running the telephone hardware or device and enables your program to interact with the telephone device in a device-independent manner.

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

XTEL applications use *data streams* to send and receive data. The Solaris Teleservices platform maintains control of the use of these channels since multiple applications can access and use them. Data streams can transport various kinds of data such as voice, video, and fax.

XTEL API Characteristics

The XTEL 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.

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 is of interest to your 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.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

XTEL Events

XTEL applications are driven by events. Events, as mentioned previously, are messages passed by the provider to your application through the notification methods. Events identify all significant XTEL occurrences such as a the state of a call or the network. You can use events to monitor the current state of a call. For instance, an application can be notified if a call is active, on hold, disconnected, or dead. Each object receives events through its notification methods.

Event Registration

In order for notification methods to receive events, you must register the events that you are interested in receiving. This is done with the `XtelProvider::listen()` method. You later choose to disregard certain events by using the `XtelProvider::ignore()` method. These methods are described in Table 3-9 on page 28.

XTEL Object Methods

XTEL classes provide virtual functions called *command methods* and *notification methods*. These standard methods generate basic call-control requests that are sent to the Teleservices platform. Depending on the available telephone resources, the system may or may not fulfill a request.

Command Methods

Each XTEL messaging class has a unique set of command methods that issues command requests to the provider for certain actions to be performed. An application invokes these methods to manipulate an object. For example, an application program that needs to put a call on hold uses the XtelCall object's `hold()` command method to change the status of the XtelCall object toput the call on hold.

Notification Methods

Objects receive events from the provider through notification methods. The events notify your application of errors or changes of state on a given call. To handle an event, you must override the notification method of a class. In

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

overriding the notification method, you change the behavior of the class so that it can perform distinct tasks. For example, you can subclass the XtelCall class to answer a call, play a greeting, and record a message; this behavior imitates an answering machine.

Overview of XTEL Classes

The XTEL API features a set of abstract data types called *classes*. The class of an object determines the methods that can be applied to it. There are two distinct base XTEL classes: messaging classes and utility classes. These include:

- Messaging Classes
  XtelProvider, XtelCall, and XtelMonitor.
- Utility Classes
  XtelCallState, XtelKVList, XtelByteArray, XtelString, IOHandler, and Dispatcher.

Each XTEL class is an abstraction of a telephony resource. Each object is designed to perform a specific telephony function. For instance, an XtelProvider object represents an active telephone device that enables a system to make and receive calls. An XtelCall object represents a call in progress, and an XtelMonitor object continuously updates and monitors the current state of a call. You can create new classes by customizing the behavior of the base XTEL messaging classes.

You can instantiate objects from any of these classes, but more likely you will subclass them to create objects with greater functionality. Further information about each class type is provided in subsequent chapters.

XtelProvider

An XtelProvider 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 XtelProvider Objects” on page 27.

XtelCall

An XtelCall represents a call in progress and provide the access method to the call’s data stream. XtelCall objects can set up, answer, release, transfer, hold, conference, and drop calls. See “Using XtelCall Objects” on page 30.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

XtelMonitor

An XtelMonitor enable applications to monitor a specific event on all calls. The XtelMonitor object allows an application to monitor all call events on a specific call, which it may or may not own.

XtelMonitor objects cannot perform call control functions. When an XtelMonitor object is created, it is automatically registered to receive all events on a specific call. See “Using XtelMonitor Objects” on page 34.

XtelCallState

The XtelCallState contains all the specific 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 XtelCallState object for immediate status only. XtelCallState objects can not be copied and saved. XtelMonitor 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 CallState object methods return state values only. CallState objects do not control calls. See “Using XtelCallState Objects” on page 26.

XtelKVList

An XtelKVList stores a key-value pair list. Keys must be of type XtelString. Values can be of type int or XtelString. The XtelKVList object is often used to information that may not conform to the standard XTEL interface. This object is used to pass arguments for extension features in the provider object. See “Using XtelKVList Objects” on page 20.

XtelByteArray

An XtelByteArray is a reference-counted array of bytes. The reference count allows for efficient use of memory because a single XtelByteArray 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 XtelByteArray Objects” on page 20.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

XtelString

An XtelString contains a null-terminated string of characters. Like the XtelByteArray object, it is reference-counted to ease memory management when strings are passed by reference between other objects. See "Using XtelString Objects" on page 20.

Dispatcher Object

The Dispatcher object detects new data on multiple UNIX file descriptors and dispatches the data to the appropriate input and output handlers. See "Using Dispatcher Objects" on page 24.

IOHandler Object

The IOHandler object is a simple object that gets called by the Dispatcher when data is available on a UNIX file descriptor. It stores pointers to three callback functions: `readready()`, `writeready()`, and `Exception()`. If you are familiar with the Interviews architecture, this mechanism of handling I/O is similar. See "Using IOHandler Objects" on page 25.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Getting Started With XTEL Programming

This chapter explains how to compile an XTEL application program. and how to make and run example programs supplied with the XTEL libraries.

Compiling XTEL Applications

Before compiling your applications, be sure that the SUNWxtel package has been installed (check that the `/opt/SUNWxtel/lib` directory exists). Your applications will need to link with two libraries in that directory: `libxtel` and `libxtelutil`.

Creating A Makefile

A makefile file greatly automates and eases the compilation and linking process for you. Critical to any XTEL makefile is the `XTELHOME` parameter. It points to the directory containing the SUNWxtel package. From that reference directory, libary and header files can be found.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Code Example 2-1 lists the makefile file for the example programs provided in the SUNWxtel package. The `XTELHOME` parameter is set to the `/opt/SUNWxtel` directory where the package was installed. From `XTELHOME`, the location for the libary and header files are derived and defined for the `INCLUDES`, `LDFLAGS`, and `LDLIBS` compiler flags.

Code Example 2-1 Makefile Example

```
#
# @(#)Makefile.demo1.1093/02/23 SMI
#

# Parameters.

XTELHOME=/opt/SUNWxtel
XTELDEMO=/opt/SUNWxtel/srcXTELHOME=/opt/SUNWxtel


# Compiler flags.

INCLUDES += -I.. -I$(XTELHOME)/include -I/usr/demo/SOUND/include

CPPFLAGS +=
LDFLAGS += -L$(XTELDEMO)/datapump -L/usr/lib -L$(OPENWINHOME)/lib \
        -L$(XTELHOME)/lib -L/usr/demo/SOUND/lib

LDLIBS += -lxtel -lxtelutil -lnsl \
        -lxview -lolgx -lX11 -lintl -laudio


DEPENDS = \
#       msgcall.h \
        voicecall.h

.KEEP_STATE:
.INIT: $(DEPENDS)

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


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Code Example 2-1 Makefile Example

```
# program rules

LINK.cc = \
        LD_RUN_PATH=$(XTELHOME)/lib:$(OPENWINHOME)/lib; export LD_RUN_PATH; \
        $(CCC) $(CCFLAGS) $(CPPFLAGS) $(LDFLAGS)

.cc.o:
        CC $(INCLUDES) -c $*.cc

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)

datapump.lib:
        -@if [ ! -f ../datapump/libdatapump.a ] ; then \
        echo "Building datapump library:"; \
        (cd ../datapump; make) \
        fi

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

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

How the XTEL Programming Interface Works

In order to receive or send a call, your application must:

1. **Create XtelProvider and XtelCall objects**

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

2. **Subclass the notification methods to receive events**
3. **Be ready to respond to significant events by using command methods and by creating new objects.**

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

Programming Tips

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 subclass three different XtelCall objects and implement the three behaviors separately. Then when a provider object gets the incoming call event, it simply creates the XtelCall object with the appropriate behavior.

In the following example program, an XtelCall object, `MyCall`, is created that sends an outgoing call using a telephone number taken from the command line.

Example Program outcall.cc

The following example program, `outcall.cc`, shows how you can make an outgoing call by creating provider and call objects. This program creates a provider object to get associated with a device, creates a call object to own and send the call, 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 39. For now, try compiling the program to ensure that your programming environment is set up appropriately.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Code Example 2-2 Creating an Outgoing Call, outcall.cc *(1 of 3)*

```
/* Copyright (c) 1992 by Sun Microsystems, Inc. */
#ident"@(#)outcall.cc1.1692/12/17 SMI"

#include <xtel/xtel.h>
#include <xtel/xtelprovider.h>
#include <xtel/xtelcall.h>
#include <xtel/xtelcallstate.h>
#include <xtel/dispatcher.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <strings.h>

#include "voicecall.h"


class MyProvider : public XtelProvider {
public:
    MyProvider(Error* err, XtelString pname) : XtelProvider(err, pname) {}
    virtual void activated(XtelKVList&);
    virtual void deactivated(XtelKVList&);
};

class MyCall : public DirectAudioCall {
public:
    MyCall(XtelProvider& xp, XtelAddress rn) : DirectAudioCall(xp, rn) {}
    virtual void            event(CallEvent, XtelKVList&);
};

char    Number[80];
MyCall* currentCall=0;

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

    // Make out going call
    if (currentCall)
    delete currentCall;
    currentCall = new MyCall(*this, Number);
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Code Example 2-2 Creating an Outgoing Call, outcall.cc *(2 of 3)*

```
}

void
MyProvider::deactivated(XtelKVList&)
{
    fprintf(stderr, "Provider died.\n");
}

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

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

void
sigint(int)
{
    exit(1);
}

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

    signal(SIGINT, sigint);

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

    if (argc == 3) {
    pvname = argv[2];
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Code Example 2-2 Creating an Outgoing Call, outcall.cc *(3 of 3)*

```
    } else {
    pvname = NULL;
    }

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

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


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

XTEL Classes 3

By using XTEL classes, your applications can create objects that act as building blocks to configure and use teleservices resources. XTEL classes offer methods that perform basic telephone operations , such as making, receiving, holding, and transfering calls. After an object completes its function, the application 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 XTEL provides. For introductory information on using these classes, see "Getting Started With XTEL Programming" on page 7. See "Creating XTEL Applications" on page 37 for more detailed information on writing specific XTEL programs.

This chapter describes each XTEL class and its methods in detail. But first, you need to understand the XTEL events that can be sent through an object's notification methods and the XTEL library global types.

Building Blocks

XTEL uses several fundamental data structures and data types to pass status and messages between objects and to pass information between processes. These data types and structures include:

- XtelCallReference values
- Events
- XtelKVList structures

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

- XtelCallState structures

You should familiarize yourself with these basic buiding blocks so that you can better understand how the object methods use them.

Using XtelCallReference Values

An XtelCallReference is a value that uniquely identifies a call that exists on a host machine. The XtelCallReference value is contained in another data structure called an XtelCallState, which maintains state information about a call.

An XtelCallReference is useful for manipulating a call. XtelCallReference values are typically passed between objects that need to act on a call. Furthermore, since the value is unique to a host, an XtelCallReference can also be passed between applications (processes) using any standard form of UNIX interprocess communication.

Events

Because XTEL applications are event driven, you must decide which events are of interest and then subclass the notification methods that will handle the event when it arrives.

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

After registering for the events, you need to either override (subclss) the relevant methods with its own event handler, or do nothing and accept the default behavior. For methods that must be subclassed because its default behavior does not adequately ha

The XtelCall object is the only object that uses events. An application can use an XtelCall object to register for events on a particular call.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Events that show the current state of a call are listed in Table 3-1 and Table 3-2.

Table 3-1 CallState Events (from *xtel.h*)

Event	Description
UNKNOWN_MAJOR_STATE	The major state of this call is unknown
CREATED_CALL	This call is created.
INCOMING_CALL	This call is an incoming call.
ACTIVE_CALL	This call is currently active.
HELD_CALL	This call is on hold.
DISCONNECTED_CALL	This call has been disconnected

Table 3-2 AdvisoryState Events (from *xtel.h*)

Event	Description
PROCEEDING	A complete address has been accepted.
ALERTING	The other end of the call is ringing.
PROGRESSING	The call is in progress.
NETWORK_BUSY	The local network is busy and your call cannot be made.
BUSY	Your call has reached the other end, but it is busy and cannot respond.
END_TO_END	The other end of the call has been picked up, and the connection has been accepted.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Table 3-3 list CallEvent events. These are events your application may want to listen for and respond to when they occur.

Table 3-3 CallEvent Events (from *xtel.h*)

Event	Descriptions
Advisory Events	Advisory events are network dependent and are not guaranteed to be relayed to your application, hence they are advisory. Alhough you can register to listen to these events and react accordingly, your applications should not rely on these advisory events to function properly.
PROCEEDING_EVENT	An address has been accepted.
ALERTING_EVENT	The other end of the call is ringing.
PROGRESSING_EVENT	A call is in progress.
NETWORK_BUSY_EVENT	The network is currently busy.
BUSY_EVENT	The other end of the call is busy.
END_TO_END_EVENT	The receiver has been picked up, and the connection has been accepted.
Call Instantiation Events	Call instantiation events occur when a call is first created, changes owner, or becomes invalid (disconnected). These events can only be seen by the XtelProvider object, with the exception of the CHANGED_OWNER_EVENT, which XtelMonitor objects can also receive.
CHANGED_OWNER_EVENT	The call now has a new owner. This event causes the assoicated call object to be deactivated.
CREATED__CALL_EVENT	A call has been created and activated().
INVALID_CALL_EVENT	A call is no longer valid and has been deactivated().
CallState Events	CallState events can be received by all XtelCall objects and are guaranteed to be sent when the defined events occur.
ACTIVE_CALL_EVENT	A call has been established and its data stream is available for use and configuration.
HELD_CALL_EVENT	A call has been put on hold (by the switch). The call remains established, but its data stream is not available for use or configuration.
DISCONNECTED_CALL_EVENT	A call is no longer established and its data stream is no longer available.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Table 3-3 CallEvent Events (from *xtel.h*) (Continued)

Event	Descriptions
TRANSFER_EVENT	Acall has been transfered. This event is received by the call object that transfered the call.
CONFERENCE_EVENT	Acall has been conferenced.
DROP_EVENT	A call has been dropped.
INFO_EVENT	The call object has received an Info packet from the switch.

XTEL Object Relationships

Messaging objects (XtelProvider, XtelCall, and XtelMonitor) share a common base class called XtelObject. From XtelObject, the three messaging classes are derived. The remaining utility classes are standalone and are used as needed. Some XTEL objects, however, do interact closely. For instance, XtelProvider and XtelCall objects are the only objects that can create XtelCallState objects. XtelCallState objects do not control calls, but contain specific state information that pertains to a call.

XtelCallState objects provide information on a specific instance of a call. XtelMonitor objects allow an application to continuously access current information on a specific call.

Object Activation and Deactivation

The messaging classes have two *object state* functions in common: `activated()` and `deactivated()`. These functions are notification methods that you must override to perform initialization when the object is activated and cleanup when the object is deactivated.

XTEL Classes

As introduced in previous chapters, the following XTEL classes are available:

- Messaging Classes
  - XtelProvider
  - XtelCall
  - XtelMonitor

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

- Utility Classes
  - XtelKVList
  - XtelByteArray
  - XtelByteString
  - XtelCallState
  - IOHandler
  - Dispatcher

The following sections describe each class's command and notification methods.

When reading about these methods, you may notice the correspondence between certain command and notification methods. For example, `listen()` and `listenReply()`, or `getCallState()` and `getCallStateReply()`, are paired command and notification methods. The Reply suffix indicates the command-notification relationship. For example, when a `listen()` command is given, XTEL invokes the `listenReply()` to return the result and status of the command method. Other methods follow this same convention.

Using XtelByteArray Objects

<to be documented>

Using XtelString Objects

<to be documented>

Using XtelKVList Objects

XtelKVList objects are used to pass provider-specific information to certain methods. However, using too many provider-specific arguments in an XtelKVList sacrafices the some portability aspects of your applications.

An XtelKVList object is a first-in, first-out ordered list of *key* and *value* pairs as shown in figure 3-1. An XtelKVList is also reference counted to better manage memory. A *key* is any null-terminated string while the corresponding *value* can be of type `u_long`, `char*` (null terminated), `XtelString&`, or `XtelByteArray&`.

Some characteristics of an XtelKVList object are:

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

- An XtelKVList is ordered and traversed sequentially. An internal pointer points to the current position in the list.
- XtelKVLists are refrence counted to relieve you of memory management chores when a list is shared between several objects.
- 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.
- The order that key-value pairs are added to the list is the same order in which they are read out. That is, in a first-in, first-out manner.
- 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 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 3rd 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).

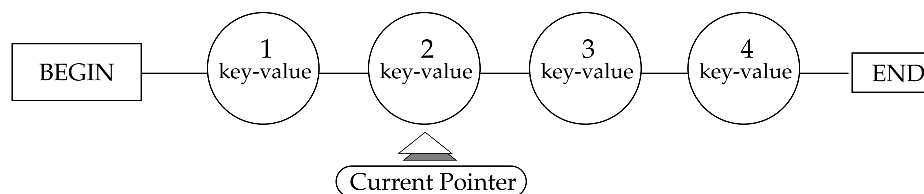


Figure 3-1 XtelKVList Structure

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Use the `XtelKVList()` constructor to create an XtelKVList object. The XtelKVList class offers the command methods listed in Table 3-4.

Table 3-4 XtelKVList Class Command Methods

Return Code Type	Method	Description
`boolean_t`	`add(const XtelString& key, u_long val)`	Add the specified key-value pair to the end of the list. If successful, `B_TRUE` is returned; otherwise `B_FALSE` is returned.
`boolean_t`	`add(const XtelString& key, const XtelString& val)`	Add the key-value pair to the end of the list. `B_TRUE` is return on success; otherwise `B_FALSE` is returned.*
`boolean_t`	`add(const XtelString& key, const XtelByteArray& val)`	Add the key-value pair to the end of the list. `B_TRUE` is returned on success or `B_FALSE` on failure.*
`boolean_t`	`add(const XtelString& key, const char* val)`	Add the key-value pair to the end of the list. Note `val` is a constant pointer to a `char`. `B_TRUE` is returned on success or `B_FALSE` on failure. *__NOTE:__ `add()` copies `XtelString` and `XtelByteArray` values so that the list contains copies, not references, of those values.
`boolean_t`	`remove()`	Remove the current key-value pair on the list.
`boolean_t`	`get(u_long& val)`	Get the value of the current key-value pair. If the value is a `u_long`, `val` is a reference 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(XtelString& val)`	Get the value of the current key-value pair. If the value is an `XtelString`, `val` is a reference 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(XtelByteArray& val)`	Get the value of the current key-value pair. If the value is an `XtelByteArray`, `val` is a reference to the value and `B_TRUE` is returned. If the value is of another type, `val` is undefined and `B_FALSE` is returned.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Table 3-4 XtelKVList Class Command Methods

Return Code Type	Method	Description
boolean_t	key(XtelString& key)	Gets the key of the current key-value pair. If there is no current pair, key is undefined and B_FALSE is returned. Otherwise, key is a reference to the key and B_TRUE is returned.
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	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.
boolean_t	first(const XtelString& 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 value matches the key argument. The return code is the return code from next().
boolean_t	next(const XtelString& key)	Advances the current pointer to the next key-value pair that has a key value 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 after the last pair in the list.
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() fails after reset() unless a next() is first performed.

An application can make a new reference to XtelKVList information with the following XtelKVList methods:

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


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Using the "=" operator copies a reference to an XtelKVList. It does not copy the data. Therefore changes that you make afftect the same list

To actually copy data from an XtelKVList, you should construct a new XtelKVList and copy the key-value pairs using `get()`, `key()`, and `add()`.

Using Dispatcher Objects

The Dispatcher object detects new data on multiple UNIX file descriptors and dispatches the data to the appropriate input and output handlers. An application only needs one instance of a dispatcher. The static member function, `Dispatcher::instance()`, is available to create an instance of the dispatcher, if necessary, and then return a reference to it.

Note – IOHandler return values affect the behavior of the dispatcher.

If an IOHandler returns a negative value, the dispatcher will initiate the `unlink()` command and ignore the file descriptor. If a positive value is returned, the dispatcher will mark the file descriptor as ready and go through the dispatch loop again. If zero is returned, the dispatcher assumes the callback is finished with the file descriptor, and continues normally.

The `Dispatcher()` constructor creates a Dispatcher object, and the `~Dispatcher()` destructor deletes the object.

Table 3-5 shows the Dispatcher object functions.

Table 3-5 Dispatcher Object functions

Type	Method	Description
`virtual void`	`link(int fd, DispatcherMask, IOHandler*)`	`Attach IOHandler`
`virtual IOHandler*`	`handler( int fd, DispatcherMask)`	`Return IOHandler`
`virtual void`	`unlink(int fd)`	`Detach IOHandler`

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Table 3-5 Dispatcher Object functions

Type	Method	Description
`virtual void`	`startTimer( long sec, long usec, IOHandler*)`	`Start timer`
`virtual void`	`stopTimer(IOHandler*)`	`Stop timer`
`virtual boolean_t`	`setReady( int fd, DispatcherMask)`	
`virtual void`	`dispatch()`	
`virtual boolean_t`	`dispatch( long& sec, long& usec)`	
`static Dispatcher&`	`instance()`	
`static void`	`instance(Dispatcher*)`	

Using IOHandler Objects

The IOHandler object helps you to read and write the input and output functions of a UNIX file descriptor. This object is used with the dispatcher which is described in “Using Dispatcher Objects” on page 24.

The `IOHandler()` constructor creates the IOHandler object. This object consists of callback functions: `inputReady()`, `outputReady()`, and `exceptionRaised()`. The `timerExpired()` function is called when a timer started with the dispatcher has expired.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Table 3-6 shows the IOHandler object functions.

Table 3-6 IOHandler Object functions

Type	Method
`virtual int`	`inputReady(int fd)`
`virtual int`	`outputReady(int fd)`
`virtual int`	`exceptionRaised(int fd)`
`virtual void`	`timerExpired(long sec, long usec)`

Using XtelCallState Objects

An XtelCallState object is different from the other objects in that it is not created directly; it does not have a public constructor. Instead it is created by the XtelProvider and XtelCall objects as needed. XtelCallState objects contain access methods with state information that pertain to a particular instance of a call. These 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 XtelMonitor object to get continuous updates on a call's state.

The XtelCallState class offers methods that return all the important call information. Applications can access this information by calling the XtelCallState command methods listed in Table 3-7.

Table 3-7 XtelCallState Class Command Methods

Type	Method	Description
`class XtelProvider&`	`provider()`	Get reference to provider object
`CallState`	`state()`	Get call's connection state
`boolean_t`	`isSet(AdvisoryState)`	Verify setting of AdvisoryState.
`XtelAddress&`	`localNumber()`	Get local number
`XtelAddress&`	`remoteNumber()`	Get remote number
`XtelService`	`service()`	Get type of call service
`time_t`	`createTime()`	
`boolean_t`	`isOwner()`	True if you own the call

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Table 3-7 XtelCallState Class Command Methods

Type	Method	Description
`boolean_t`	`isIncoming()`	Get the direction of the call
`boolean_t`	`isClaimable()`	Find out if call can be claimed
`XtelCallReference`	`callReference()`	Get a call's unique identifier

The `callReference()` method returns a string that identifies a call in the system. An application can pass this function to another application in order to exchange information about the call.

Table 3-8 shows this object's global command method.

Table 3-8 XtelCallState Global Command Method (from *xtelcallstate.h*)

Type	Method	Description
`static XtelString`	`providerName(XtelCallReference&)`	Get the name of this provider.

Using XtelProvider Objects

An XtelProvider object represents a connection between a network and a telephone device. It is essentially a phone line that enables your phone to make and receive calls. XtelProvider objects are required for both the creation and reception of connections. Table 3-9 lists all of the XtelProvider class command methods.

You create XtelProvider objects by calling the XtelProvider constructor, `XtelProvider()` which initializes the XtelProvider class. The following code shows provider object creation:

```
XtelProvider* p = new XtelProvider(Error* err, XtelString pname);
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Table 3-9 XtelProvider Class Command Methods (from *xtelprovider.h*)

Type	Command Method	Description
`virtual XtelString`	`name()`	Get primary alias for this provider.
`virtual Error`	`forward( XtelAddress local, XtelAddress remote, XtelKVList& args = *Xtel::Null_KVList)`	Forward a call. Not implemented.
`virtual Error`	`unforward( XtelAddress address, XtelKVList& args = *Xtel::Null_KVList)`	Turn off call-forwarding. Not implemented.
`virtual Error`	`listAllCalls()`	List active calls on this provider.
`virtual Error`	`enableOfferEvent(boolean_t on)`	Get an offered event.
`virtual Error`	`listen(CallEvent listenFor)`	Notify or listen for named events.
`virtual Error`	`ignore(CallEvent toIgnore)`	Stop listening for named events.
`virtual Error`	`sendMessage(XtelKVList& args = *Xtel::Null_KVList)`	Sends arguments in the XtelKVList to the provider. This is an extension mechanism used to activate provider-specific features.
`virtual Error`	`getCallState(XtelCallReference& ref)`	Gets the state of a call, which is specified by the XtelCallReference value.

The application can use the notification methods in Table 3-10 to register this provider to receive related events.

Table 3-10 XtelProvider Class Notification Methods (from *xtelprovider.h)*

Type	Notification Method	Description
`virtual void`	`activated(XtelKVList& args)`	This object has been activated and can receive messages.
`virtual void`	`deactivated(XtelKVList& args)`	This object has been deactivated and can no longer receive messages.
`virtual void`	`forwardReply( XtelAddress local, XtelAddress remote, XtelKVList& args)`	Reply to forward() command method.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Table 3-10 XtelProvider Class Notification Methods (from *xtelprovider.h)*

Type	Notification Method	Description

`virtual void`	`unforwardReply(`
	`XtelAddress address,`
	`XtelKVList& args)`
		Reply to unforward() command method.

`virtual void`	`listAllCallsReply(XtelKVList& args)`	Get list of active calls
`virtual void`	`enableOfferEventReply(boolean_t on)`	Random call state change
`virtual void`	`listenReply(CallEvent listeningTo)`	Listen for specific CallEvents
`virtual void`	`ignoreReply(CallEvent ignoring)`	Ignore specific CallEvents

`virtual void`	`getCallStateReply(`
	`XtelCallState& state,`
	`XtelKVList& args)`
		Returns the call state of a call.

`virtual void`	`callOfferEvent(`
	`XtelCallState& call,`
	`XtelKVList& args)`
		A call is available to be claimed.

`virtual void`	`callevent(`
	`XtelCallState& call,`
	`CallEvent event,`
	`XtelKVList& args)`
		An event has occured on a given call.

`virtual void`	`error(`
	`Request command,`
	`Xtel::Error err,`
	`XtelKVList& args)`
		An error occurred in a given command request. The values for `Request` and `Error` are shown in Table 3-11

`virtual void`	`overloadWarning(char* method)`	This method is called when a notification method has not been subclassed (overloaded). This warning occurs because the default behavior of the method does not appropriately handle an event.When invoked, overloadWarning() prints the message: *`object_name`*`::`*`function_name`*`()` `invoked but not overridden,` `obj=`*this_address*.
`virtual void`	`receiveMessage(XtelKVList& args)`	The provider has received the message sent by `sendMessage()`.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

The `Request` and `Error` parameters in Table 3-10 are enumerated types, and can have the values listed in Table 3-11. The `Request` values identify the offending request and correspond to the respective command method names. The `Error` values Indicates the type or cause of an error.

Table 3-11 XtelProvider Request and Error Values (from *xtelprovider.h*)

Type	Possible Values	Description
enum Request	CREATE_PROVIDER, LISTEN, IGNORE, LIST_CALLS, FORWARD, UNFORWARD, SEND_MESSAGE, GET_CALL_STATE,	Request values correspond to the offending command method name.
	UNKNOWN_REQUEST	The provider received an unrecognizd request.
enum Error	SUCCESS	No error.
	PROVIDER_CONNECT_ERROR	Could not pass messages to the provider. This indicates a configuration or incomplete startup problem.
	PROVIDER_NOT_FOUND_ERROR	Messages could be passed but the intended provider was not found. This indicates that the provider has not been started with *xteltool*(1).
	FAILURE	Command method failed.

Using XtelCall Objects

An XtelCall object is the fundamental XTEL object. It represents a call in progress, and provides the access method to the call's data stream. XtelCall object command methods generate requests and responses for call control functions such as setting up, answering and releasing calls. They can also transfer, hold, unhold, conference and drop calls.

An XtelCall object's notification methods process indications from incoming calls and calls in progress. They also confirm requests sent by the object's command methods and detect DTMF (touch tones) and silence from data streams.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

XtelCall objects are the only objects that can be owned. The specific process that creates this object is the owner of that call, and has permission to control the call. The owner of the call object is automatically registered to receive all call events pertaining to that call. The owner of a call can offer the call back to all processes. The XtelCall base class offers the command methods listed in Table 3-12.

Applications can use the following constructors to create XtelCall objects:

```
XtelCall(XtelCall&)
XtelCall(XtelCallState&)
XtelCall(XtelProvider&, XtelKVList& args = *Xtel::Null_KVList)
```

Table 3-12 XtelCall Class Command Methods (from *xtelcall.h*)

Type	Command Method	Description
virtual XtelCallState&	callState()	Get reference to CallState object.
virtual Error	outgoing(XtelAddress local, XtelAddress remote, XtelService svc, XtelKVList& args = *Xtel::Null_KVList)	Dial a number to create an outgoing call.
virtual Error	answer(XtelKVList& args = *Xtel::Null_KVList)	Answer, or establish connection with an incoming call.
virtual Error	disconnect(XtelKVList& args = *Xtel::Null_KVList)	Hang up a call.
virtual Error	hold(XtelKVList& args = *Xtel::Null_KVList)	Put a call on hold.
virtual Error	unhold(XtelKVList& args = *Xtel::Null_KVList)	Take a call off hold.
virtual Error	transfer(XtelCallState& state, XtelKVList& args = *Xtel::Null_KVList)	Transfer this call to the specified CallState.
virtual Error	conference(XtelCallState& state, XtelKVList& args = *Xtel::Null_KVList)	Conference in another call.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Table 3-12 XtelCall Class Command Methods (from *xtelcall.h*)

Type	Command Method	Description
`virtual Error`	`drop(XtelKVList& args = *Xtel::Null_KVList)`	Drop last conferenced call.
`virtual Error`	`offer(XtelKVList& args = *Xtel::Null_KVList)`	Offer ownership of a call to another application using existing attributes.
`virtual Error`	`offer(XtelAddress remote, XtelAddress local, XtelService svc, XtelKVList& args = *Xtel::Null_KVList`	Offer ownership of the call to other applications using newly specified attributes.
`virtual Error`	`sendMessage( XtelKVList& args = *Xtel::Null_KVList)`	Send an extension message.
`virtual int`	`configureDataStream( XtelKVList& args).`	Configure or open a call's data stream.
`virtual Error`	`dtmfGenerate( XtelString& digits, XtelKVList& args = *Xtel::Null_KVList)`	Generate one or more DTMF tones.
`XtelString`	`string(Request req)`	Convert enumerated request to an XtelString.

An application can subclass the XtelCall notification methods in Table 3-13 to receive call events and errors.

Table 3-13 XtelCall Class Notification Methods (from *xtelcall.h*)

Type	Method	Description
`virtual void`	`configureDataStreamEvent( XtelKVList& current_configuration)`	Configure data stream events.
`virtual void`	`event(CallEvent event, XtelKVList& args)`	Notification of an event.
`virtual void`	`dtmfEvent(char value, XtelKVList&)`	Notification of a DTMF event.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Table 3-13 XtelCall Class Notification Methods (from *xtelcall.h*)

Type	Method	Description
`virtual void`	`error(Request, Xtel::Error, XtelKVList& args)`	Notification of an error.
`virtual void`	`receiveMessage(XtelKVList& args)`	Received an extension message.
`virtual void`	`overloadWarning(char* method)`	This method is a programming aid.

The XtelCall class notification methods include `Request` and `Error` parameters that are enumerated types, and can have the values listed in Table 3-14:

Table 3-14 XtelCall Request and Error Values (from *xtelcall.h*)

Parameter	Possible Values
Request	UNKNOWN REQUEST CREATE_CALL OUTGOING ANSWER DISCONNECT HOLD UNHOLD TRANSFER, CONFERENCE DROP OFFER CONFIGURE_DATA_STREAM SEND_MESSAGE DTMF_DETECT DTMF_GENERATE
Error	SUCCESS INVALID_CALL_ERROR FAILURE

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Using XtelMonitor Objects

The `XtelMonitor(XtelCallState&)` constructor creates a monitor object. The object is primarily used for logging and tracing call notification and state information and cannot perform call control functions. When a monitor object is created, it is automatically registered to receive all events.

The destructor, `virtual ~XtelMonitor()` destroys monitor objects.

Table 3-15 shows the XtelMonitor object command methods.

Table 3-15 XtelMonitor Command Method

Type	Method	Description
`XtelCallState&`	`callState()`	`Access an XtelCallState object.`

Table 3-16 shows the XtelMonitor object's state methods

Table 3-16 XtelMonitor Object State Methods

Type	Method	Description
`virtual void`	`activated(XtelKVList&)`	
`virtual void`	`deactivated(XtelKVList&)`	

Table 3-17 shows the XtelMonitor object notification methods.

Table 3-17 XtelMonitor Object Notification Methods

Type	Method	Description
`virtual void`	`event(CallEvent eventcall, XtelKVList& args)`	
`virtual void`	`error(Request command, Xtel::Error err, XtelKVList& args)`	

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

The `Request` parameter is an enumerated type, and can have the following values:

Table 3-18 Request, Error, and Event Parameter values:

Type	Parameter	Arguments	Description
enum	Request	UNKNOWN REQUEST CREATE_MONITOR	Identify the type of failed request.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Creating XTEL Applications

The XTEL interface insulates you from the intricacies of the Solaris Teleservices platform and the telephone hardware installed on your system. However, to design an XTEL application, you must be aware of the capabilities of your telephone environment. Your XTEL environment should be set up so that you can run any of the example programs discussed in this chapter. Your system should also be configured with the appropriate Teleservices providers for the telephone hardware attached to your system, Please consult your system administrator about your system configuration.

This chapter explains how you can write XTEL applications. For a complete description of each XTEL object, see “XTEL Classes” on page 19. For introductory information on writing XTEL applications, see “Getting Started With XTEL Programming” on page 7.

Creating a Basic XTEL Application

To create any XTEL applications requires the following basic steps:

1. **Include XTEL-specific header files in your source.**
2. **Specialize at least one messaging-object class to create your own classes.**
3. **Subclass the class methods to handle events of interest.**

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Header Files

Several XTEL header files are available for use in Teleservices applications. Some are optional, such as `xtel/xteldb.h`, while others, such as `xtel/xtel.h` are mandatory. Table 4-1 lists the header files.

Table 4-1 XTEL Header Files

Header File	Description
xtel/bytearray.h	Defines XtelByteArray class.
xtel/dispatcher.h	Defines Dispatcher class.
xtel/iohandler.h	Defines IOHandler class.
xtel/kvlist.h	Define XtelKVList class.
xtel/types.h	Defines simple types such as NULL, nil, true, and false. Also includes the standard <sys/types.h> header file.
xtel/xtel.h	Defines fundamental XTEL data types and structures, such as global symbols, event types, and error types.
xtel/xtelcall.h	Defines the XtelCall class, request types, and request error types.
xtel/xtelcallstate.h	Defines XtelCallState class.
xtel/xteldb.h	Declares database query functions.
xtel/xtelmonitor.h	Defines XtelMonitor class.
xtel/xtelprovider.h	Defines XtelProvider class.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Subclassing XTEL Classes

To design your own classes within an application, you subclass behavior from an Xtel object. For instance, the following code defines a class that is similar to the XtelProvider object. The new provider class, called `MyProvider`, will be notified when it is ready to send and receive messages.

```
class MyProvider : public XtelProvider {
public:
MyProvider(Error* err, XtelString pname):XtelProvider(err,pname)
{}
    virtual void activated(XtelKVList&);
    virtual void deactivated(XtelKVList&);
};
```

In the above example `MyProvider` is subclassed from XtelProvider. `MyProvider` inherits all of the XtelProvider class's functionality. The subclass function declarations specify the behavior of the new class, `MyProvider`.

In addition, the `activated()` and `deactivated()` state methods are defined. You must define these methods for every messaging object you subclass 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 an XTEL events, the class you create to receive that event must subclass one of the XTEL messaging classes: XtelProvider, XtelCall, or XtelMonitor.

Program Example outcall.cc

To illustrate the basic code used in an XTEL application, we will step through the program *outcall.cc* as an example; presents a listing of the program. The *outcall.cc* program creates an outgoing call by subclassing two XTEL classes, XtelProvider and XtelCall, into subclasses called `MyProvider` and `MyCall`, respectively.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Code Example 4-1 Listing of outcall.cc

```
#include <xtel/xtel.h>
#include <xtel/xtelprovider.h>
#include <xtel/xtelcall.h>
#include <xtel/xtelcallstate.h>
#include <xtel/dispatcher.h>
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <strings.h>

#include "voicecall.h"

class myProvider : public XtelProvider {
public:
    myProvider(Error* err, XtelString pname) : XtelProvider(err, pname) {}
    virtual void activated(XtelKVList&);
    virtual void deactivated(XtelKVList&);
};

class myCall : public DirectAudioCall {
public:
    myCall(XtelProvider& xp, XtelAddress rn) : DirectAudioCall(xp, rn) {}
    virtual void event(CallEvent, XtelKVList&);
};

char        number[80];
myCall*     currentCall=NULL;

void myProvider::activated(XtelKVList&) {
    fprintf(stderr, "Using provider: %s\n", (name())());

    // Make out going call
    if (currentCall)
        delete currentCall;
    currentCall = new myCall(*this, Number);
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Code Example 4-1 Listing of outcall.cc

```
}

void myProvider::deactivated(XtelKVList&)
{
    fprintf(stderr, "Provider died.\n");
}

void myCall::event(CallEvent event, XtelKVList& kvl)
{
    // preserve DirectAudioCall behavior
    DirectAudioCall::event(event, kvl);

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


main(int argc, char* argv[])
{
    XtelProvider::Errorerr;
    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;
    }
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Code Example 4-1 Listing of outcall.cc

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

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

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

The first step in the program is to define the `MyProvider` and `MyCall` classes. The `MyProvider` class creates a provider, and waits for events that indicate that the provider, specified by `pname`, has been `activated()` or `deactivated()`:

```
class MyProvider : public XtelProvider {
public:
    MyProvider(Error* err, XtelString pname) :
                                XtelProvider(err, pname) {}
    virtual void activated(XtelKVList&);
    virtual void deactivated(XtelKVList&);
};
```

The `MyCall` class is subclassed from the `DirectAudioCall` class, and if you look at the `voicecall.h` file, you can see that `DirectAudioCall` is derived from the XtelCall class. Furthermore, `MyCall`’s functionality is extended by

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

defining it to receive call events through `event()`. `MyCall` also implicity inherits the behavior of the `activated()` and `deactivated()` methods from the `DirectAudioCall` class.

```
class MyCall : public DirectAudioCall {
public:
    MyCall(XtelProvider& xp, XtelAddress rn) :
                                 DirectAudioCall(xp, rn) {}
    virtual void             event(CallEvent, XtelKVList&);
};
```

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()`. This notification is accomplished by overriding the XtelProvider object's `activated()` notification function. The XTEL platform calls `MyProvider::activated()` 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(XtelKVList&)
{
    fprintf(stderr, "Using provider: %s\n", (name())());

    // Make out going 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 *xteltool*(1) configuration program. For more information on provider aliases, see "Querying the XTEL Configuration Database" on page 67.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

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

```
void
MyProvider::deactivated(XtelKVList&)
{
    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 XtelCall object's event() method; it this example, and event() takes on DirectAudioCall's functionality:

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

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


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Now that everything is defined, we introduce the `main()` function. 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 which to invoke the `outgoing()` call.

```
main(int argc, char* argv[])
{
    XtelProvider::Errorerr;
    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, XtelString(pvname));
    if (err != XtelProvider::SUCCESS) {
        fprintf(stderr, "could not connect to provider\n");
        exit(1);
    }

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


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Handling Audio

The following program, voicecall.cc, is an example of an XTEL audio implementation. This program's header file is used in all of the XTEL programming examples.

Code Example 4-2 Listing of `voicecall.cc` *(1 of 3)*

```
/* Copyright (c) 1992 by Sun Microsystems, Inc. */
#ident"@(#)voicecall.cc1.893/03/09 SMI"



#include <xtel/kvlist.h>
#include <xtel/xtelcallstate.h>
#include "voicecall.h"

DirectAudioCall::DirectAudioCall(
    XtelProvider& xtelpv,
    XtelAddress remoteNumber)
    : XtelCall(xtelpv)
{
    _remoteNumber = remoteNumber;
}


void
DirectAudioCall::activated(XtelKVList&)
{
    switch (callState().state()) {
    case CREATED_CALL:
        outgoing("", _remoteNumber, "XTEL_8KHZ_ULAW_VOICE");
    break;
    case INCOMING_CALL:
        answer();
    break;
    case ACTIVE_CALL:
        event(ACTIVE_CALL_EVENT, *Xtel::Null_KVList);
    break;
    case HELD_CALL:
        event(HELD_CALL_EVENT, *Xtel::Null_KVList);
    break;
    // unsupported by DirectAudioCall
    case DISCONNECTED_CALL:
    case INVALID_CALL:
    default:
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Code Example 4-2 Listing of `voicecall.cc` *(2 of 3)*

```
/* Copyright (c) 1992 by Sun Microsystems, Inc. */
        fprintf(stderr,
         "DirectAudioCall: given call in unsupported state\n");
    }
}

void
DirectAudioCall::deactivated(XtelKVList&)
{
    fprintf(stderr, "DirectAudioCall:: deactivated\n");
}

void
DirectAudioCall::event(CallEvent ev, XtelKVList&)
{
    XtelKVList kv_list;

    //fprintf(stderr, "DirectAudioCall: PROGRESSING = %d, ",
    // callState().isSet(PROGRESSING));
    //fprintf(stderr, "ALERTING = %d, ",
callState().isSet(ALERTING));
    //fprintf(stderr, "END_TO_END = %d\n",
callState().isSet(END_TO_END));

    switch (ev) {
    case ACTIVE_CALL_EVENT:
        configureDataStream(*Xtel::ConnectToDefaultDevice);
    break;
    case HELD_CALL_EVENT:
        configureDataStream(*Xtel::NullConfiguration);
    break;
    case ALERTING_EVENT:
    break;
    case PROCEEDING_EVENT:
    break;
    case END_TO_END_EVENT:
    break;
    case DISCONNECTED_CALL_EVENT:
    break;
    }

}

void
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Code Example 4-2 Listing of `voicecall.cc` *(3 of 3)*

```
/* Copyright (c) 1992 by Sun Microsystems, Inc. */
DirectAudioCall::configureDataStreamEvent(XtelKVList&)
{
    fprintf(stderr, "audio configuration event\n");
}


void
DirectAudioCall::receiveMessage(XtelKVList&)
{
    fprintf(stderr, "received extension message\n");
}
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

The following is the `voicecall.h` file:

Code Example 4-3 Listing of `voicecall.h`

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

#ifndef _MULTIMEDIA_VOICECALL_H
#define_MULTIMEDIA_VOICECALL_H

#ident"@(#)voicecall.h1.893/03/09 SMI"

#include <xtel/xtelcall.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>


class DirectAudioCall : public XtelCall {
public:
    DirectAudioCall(XtelProvider& xtelpv, XtelAddress
remoteNumber);
    DirectAudioCall(XtelCallState& call) : XtelCall(call) {}
    virtual void            event(CallEvent, XtelKVList&);
    virtual void            receiveMessage(XtelKVList&);
    virtual voidconfigureDataStreamEvent(XtelKVList&);
    virtual void activated(XtelKVList&);
    virtual void deactivated(XtelKVList&);
private:
    XtelAddress _remoteNumber;
};

#endif /* !_MULTIMEDIA_VOICECALL_H */
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Answering Incoming Calls

The following program, `incall.cc`, answers an incoming call. This program calls the `voicecall.h` header file explained in "Handling Audio" on page 46.

Code Example 4-4 Listing of `incall.cc` *(1 of 4)*

```
/* Copyright (c) 1992 by Sun Microsystems, Inc. */
#ident"@(#)incall.cc1.1592/12/17 SMI"

#include <xtel/xtel.h>
#include <xtel/xtelprovider.h>
#include <xtel/xtelcall.h>
#include <xtel/xtelcallstate.h>
#include <xtel/dispatcher.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

#include "voicecall.h"

DirectAudioCall*audiocall=0;

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

class MyProvider : public XtelProvider {
public:
    MyProvider(Error* err, XtelString pname) : XtelProvider(err,
pname) {}
    virtual void activated(XtelKVList&);
    virtual void deactivated(XtelKVList&);
    virtual void callOfferEvent(XtelCallState& cs, XtelKVList&);
    virtual void callevent(XtelCallState&, CallEvent,
XtelKVList&);
};


void
MyProvider::callOfferEvent(XtelCallState& call, XtelKVList&)
{
    // pick up any call offered to us
    if (audiocall)
        delete audiocall;
    audiocall = new DirectAudioCall(call);
}
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Code Example 4-4 Listing of `incall.cc` *(2 of 4)*

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

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

    // register for incoming call events
    enableOfferEvent(B_TRUE);
}

void
MyProvider::deactivated(XtelKVList&)
{
    fprintf(stderr, "Provider died.\n");
}

void
MyProvider::callevent(XtelCallState& , CallEvent ev,
XtelKVList&)
{
    // Not listening for all these events
    //
    switch (ev) {
    case PROCEEDING_EVENT:
        fprintf(stderr, "heard PROCEEDING_EVENT\n");
    break;
    case ALERTING_EVENT:
        fprintf(stderr, "heard ALERTING_EVENT\n");
    break;
    case PROGRESSING_EVENT:
        fprintf(stderr, "heard PROGRESSING_EVENT\n");
    break;
    case NETWORK_BUSY_EVENT:
        fprintf(stderr, "heard NETWORK_BUSY_EVENT\n");
    break;
    case BUSY_EVENT:
        fprintf(stderr, "heard BUSY_EVENT\n");
    break;
    case END_TO_END_EVENT:
        fprintf(stderr, "heard END_TO_END_EVENT\n");
    break;
    case CONFERENCE_EVENT:
        fprintf(stderr, "heard CONFERENCE_EVENT\n");
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Code Example 4-4 Listing of `incall.cc` *(3 of 4)*

```
/* Copyright (c) 1992 by Sun Microsystems, Inc. */
    break;
    case CHANGED_OWNER_EVENT:
        fprintf(stderr, "heard CHANGED_OWNER_EVENT\n");
    break;
    case CREATED_CALL_EVENT:
        fprintf(stderr, "heard CREATED_CALL_EVENT\n");
    break;
    case ACTIVE_CALL_EVENT:
        fprintf(stderr, "heard ACTIVE_CALL_EVENT\n");
    break;
    case HELD_CALL_EVENT:
        fprintf(stderr, "heard HELD_CALL_EVENT\n");
    break;
    case DISCONNECTED_CALL_EVENT:
        fprintf(stderr, "heard DISCONNECTED_CALL_EVENT\n");
    break;
    case INVALID_CALL_EVENT:
        fprintf(stderr, "heard INVALID_CALL_EVENT\n");
    break;
    default:
        fprintf(stderr, "bad number event!\n");
    }

}

void
sigint(int)
{
    exit(1);
}

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

    signal(SIGINT, sigint);

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


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Code Example 4-4 Listing of `incall.cc` *(4 of 4)*

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

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

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

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

Using the Dispatcher and Notifier Interfaces

The following example program shows how you can use the dispatcher.

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


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

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

    return NOTIFY_DONE;
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

```
}

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);
}
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Creating an Answering Machine

The following example code (machine.cc) creates an answering machine.

Code Example 4-5 Listing of machine.cc *(1 of 5)*

```
/* Copyright (c) 1992 by Sun Microsystems, Inc. */
#ident"@(#)machine.cc1.993/03/09 SMI"

#include <xtel/xtel.h>
#include <xtel/xtelprovider.h>
#include <xtel/xtelcall.h>
#include <xtel/xtelcallstate.h>
#include <xtel/dispatcher.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

#include "msgcall.h"

MsgCall*msgcall;

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

class MyCall : public MsgCall {
public:
    MyCall(XtelCallState& cs, char* announcement, char* message)
:
     MsgCall(cs, announcement, message) {}
    virtual void playDone();
    virtual void activated(XtelKVList&);
    virtual void event(CallEvent ev, XtelKVList&);
};

void
MyCall::activated(XtelKVList&)
{
        fprintf(stderr,
         "MyCall: activeated \n");
    switch (callState().state()) {
    case INCOMING_CALL:
        answer();
    break;
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Code Example 4-5 Listing of `machine.cc` *(2 of 5)*

```
/* Copyright (c) 1992 by Sun Microsystems, Inc. */
    case ACTIVE_CALL:
        playAnnc();
    break;
    // unsupported by MsgCall
    case CREATED_CALL:
    case HELD_CALL:
    case DISCONNECTED_CALL:
    case INVALID_CALL:
    default:
        fprintf(stderr,
         "MyCall: given call in unsupported state\n");
    }
}

void
MyCall::playDone()
{
    recordMsg();
}

void
MyCall::event(CallEvent ev, XtelKVList&)
{
    switch(ev) {
    case ACTIVE_CALL_EVENT:
        playAnnc();
    break;
    // ignored by MsgCall
    case END_TO_END_EVENT:
    case ALERTING_EVENT:
    case PROCEEDING_EVENT:
    break;
    case DISCONNECTED_CALL_EVENT:
        stopRecordMsg();

        // this code only manages one call at a time, delete
        // only call and reset global pointer
        delete this;
        msgcall = NULL;
    break;
    // unsupported by MyCall
    case HELD_CALL_EVENT:
    default:
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Code Example 4-5 Listing of `machine.cc` *(3 of 5)*

```
/* Copyright (c) 1992 by Sun Microsystems, Inc. */
        fprintf(stderr,
         "MyCall: ignoring %s Event\n", Xtel::string(ev)());
    }
}


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

class MyProvider : public XtelProvider {
public:
    MyProvider(Error* err, XtelString pname) : XtelProvider(err,
pname) {}
    virtual void activated(XtelKVList&);
    virtual void deactivated(XtelKVList&);
    virtual void callOfferEvent(XtelCallState& cs, XtelKVList&);
    virtual void callevent(XtelCallState&, CallEvent,
XtelKVList&);
};


void
MyProvider::callOfferEvent(XtelCallState& call, XtelKVList&)
{
    fprintf(stderr, "got callOfferEvent\n");

    // pick up incoming call
    if (call.state() == INCOMING_CALL) {
        fprintf(stderr, "incoming call....\n");
        msgcall = new MyCall(call,
         "/usr/demo/SOUND/sounds/train.au", "/tmp/msg.au");
    }
}

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

    fprintf(stderr, "got CREATE_PROVIDER_EVENT\n");
    // register for incoming call events
    enableOfferEvent(B_TRUE);
}
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Code Example 4-5 Listing of `machine.cc` *(4 of 5)*

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

void
MyProvider::deactivated(XtelKVList&)
{
    fprintf(stderr, "Provider died.\n");
}

void
MyProvider::callevent(XtelCallState& , CallEvent ev,
XtelKVList&)
{
    // Not currently listening for any provider events
    //
    fprintf(stderr, "MyProvider:: Ignoring %s provider event\n",
     Xtel::string(ev)());
}

void
sigint(int)
{
    fprintf(stderr, "got interrupt!\n");
    delete msgcall;
    exit(1);
}

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

    signal(SIGINT, sigint);

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

    char*       pvname;
    if (argc == 2) {
        pvname = argv[1];
    } else {
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Code Example 4-5 Listing of `machine.cc` *(5 of 5)*

```
/* Copyright (c) 1992 by Sun Microsystems, Inc. */
        pvname = NULL;
    }

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

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

The machine.cc program calls the msgcall.h header file. The msgcall.cc program sends calls to the answering machine:

Code Example 4-6 Listing of `msgcall.h` *(1 of 3)*

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

#ifndef _MULTIMEDIA_SPHCALL_H
#define_MULTIMEDIA_SPHCALL_H

#ident "@(#)msgcall.h 1.13 93/03/09 SMI"

#include <xtel/xtelcall.h>
#include <datapump/datapump.h>
#include <xtel/iohandler.h>
#include <datapump/dtmfdet.h>

class AudioWriter : public CopyWriter, public IOHandler {
public:
    AudioWriter(DataPump& audio, int file_fd, void* client_data =
NULL,
     int bufsize = 1024);
    virtual ~AudioWriter();
    virtual void donePlaying();
    virtual void* clientData();
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Code Example 4-6 Listing of `msgcall.h` *(2 of 3)*

```
protected:
    boolean_t playedZeroLengthBuffer(int fd);
    virtual void timerExpired(long sec, long usec);
private:
    int     audio_device;
    void*   client_data;
};

class DTMFHandler : public DTMFReader {
public:
    DTMFHandler(DataPump& audioPump) : DTMFReader(audioPump) {}
    virtual void detectedToneUp(char c);
    virtual void detectedToneDown(char c);
    virtual void detectedToneString(char *);
};

class MsgCall : public XtelCall {
public:
    MsgCall(XtelCallState& cs, char* announcement, char*
message);
    virtual ~MsgCall();
    virtual void activated(XtelKVList&);
    virtual void event(CallEvent ev, XtelKVList&);
    virtual void configureDataStreamEvent(XtelKVList&
newconfig);
    virtual void error(Request req, Xtel::Error err,
XtelKVList&);
    virtual int playAnnc();
    virtual void playDone();
    virtual void stopPlayAnnc();
    virtual int recordMsg();
    virtual void stopRecordMsg();
    char* messageFile();
    char* announcementFile();
private:
    virtual void start_play();
    virtual void start_record();
    int     _audioFd;
    int     _anncFd;
    int     _msgFd;
    char*   _anncAudioFile;
    char*   _msgAudioFile;
    int     verify_openDataStream();
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Code Example 4-6 Listing of `msgcall.h` *(3 of 3)*

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

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

class CallGreeting : public MsgCall {
public:
    CallGreeting(XtelCallState&, int audioFileFd);
    //CallGreeting(XtelCall&, int audioFileFd);
    virtual ~CallGreeting();

    virtual void event(CallEvent ev, XtelKVList&);
    virtual void playDone();
private:
};

#endif /* !_MULTIMEDIA_SPHCALL_H */
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Monitoring Calls

The following program uses the XtelMonitor object to monitor calls.

Code Example 4-7 Listing of `monitorcalls.cc` *(1 of 5)*

```
/* Copyright (c) 1992 by Sun Microsystems, Inc. */
#ident"@(#)monitorcalls.cc1.1593/01/27 SMI"

#include <xtel/xtel.h>
#include <xtel/xtelprovider.h>
#include <xtel/xtelmonitor.h>
#include <xtel/xtelcallstate.h>
#include <xtel/dispatcher.h>
#include <stdio.h>
#include <stdlib.h>

class MyProvider : public XtelProvider {
public:
    MyProvider(Error* err, XtelString pname) : XtelProvider(err,
pname) {}
    virtual void activated(XtelKVList&);
    virtual void deactivated(XtelKVList&);
    virtual void callOfferEvent(XtelCallState&, XtelKVList&);
    virtual void callevent(XtelCallState&, CallEvent,
XtelKVList&);
    virtual void listAllCallsReply(XtelKVList& args);
};

class MyMonitor : public XtelMonitor {
public:
    MyMonitor(XtelCallState& cs) : XtelMonitor(cs) {}
    virtual void deactivated(XtelKVList&);
    virtual void event(CallEvent, XtelKVList&);
};

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

    // get list of existing calls
    listAllCalls();
    // register for new calls
    enableOfferEvent(B_TRUE);
    listen(CREATED_CALL_EVENT);
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Code Example 4-7 Listing of `monitorcalls.cc` *(2 of 5)*

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

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

void
MyProvider::callOfferEvent(XtelCallState& cs, XtelKVList&)
{
    MyMonitor*monitor;
    monitor = new MyMonitor(cs);
    fprintf(stderr, "\tMonitoring newly offered call.\n");
}

void
MyProvider::callevent(XtelCallState& cs, CallEvent cev,
XtelKVList&)
{
    MyMonitor*monitor;

    switch (cev) {
    case CREATED_CALL_EVENT:
        monitor = new MyMonitor(cs);
        fprintf(stderr, "\tMonitoring newly created call.\n");
    break;
    };
}

void
MyProvider::listAllCallsReply(XtelKVList& call_list)
{
    u_long      long_arg;
    XtelCallState*cs;
    int         n=1;
    MyMonitor*monitor;

    call_list.reset();
    while (call_list.next(Xtel::CallList_Key)) {
        // print state of each existing call
        call_list.get(long_arg);
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Code Example 4-7 Listing of `monitorcalls.cc` *(3 of 5)*

```
/* Copyright (c) 1992 by Sun Microsystems, Inc. */
        cs = (XtelCallState*) long_arg;
        fprintf(stderr, "Call %d : state = %s\n",
         n++, Xtel::string(cs->state())());
        monitor = new MyMonitor(*cs);
    }
}

void
MyMonitor::deactivated(XtelKVList&)
{
    fprintf(stderr, "MyMonitor::deactivated\n");
    delete this;
}

void
MyMonitor::event(CallEvent ev, XtelKVList&)
{
    switch (ev) {
    case PROCEEDING_EVENT:
        fprintf(stderr, "heard PROCEEDING_EVENT\n");
    break;
    case ALERTING_EVENT:
        fprintf(stderr, "heard ALERTING_EVENT\n");
    break;
    case PROGRESSING_EVENT:
        fprintf(stderr, "heard PROGRESSING_EVENT\n");
    break;
    case NETWORK_BUSY_EVENT:
        fprintf(stderr, "heard NETWORK_BUSY_EVENT\n");
    break;
    case BUSY_EVENT:
        fprintf(stderr, "heard BUSY_EVENT\n");
    break;
    case END_TO_END_EVENT:
        fprintf(stderr, "heard END_TO_END_EVENT\n");
    break;
    case CONFERENCE_EVENT:
        fprintf(stderr, "heard CONFERENCE_EVENT\n");
    break;
    case CHANGED_OWNER_EVENT:
        fprintf(stderr, "heard CHANGED_OWNER_EVENT\n");
    break;
    case CREATED_CALL_EVENT:
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Code Example 4-7 Listing of `monitorcalls.cc` *(4 of 5)*

```
/* Copyright (c) 1992 by Sun Microsystems, Inc. */
        fprintf(stderr, "heard CREATED_CALL_EVENT\n");
    break;
    case ACTIVE_CALL_EVENT:
        fprintf(stderr, "heard ACTIVE_CALL_EVENT\n");
    break;
    case HELD_CALL_EVENT:
        fprintf(stderr, "heard HELD_CALL_EVENT\n");
    break;
    case DISCONNECTED_CALL_EVENT:
        fprintf(stderr, "heard DISCONNECTED_CALL_EVENT\n");
    break;
    default:
        fprintf(stderr, "bad number event!\n");
    }
}


main(int argc, char* argv[])
{
    XtelProvider::Errorerr;
    MyProvider*Pv;
    Dispatcher& d = Dispatcher::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::SUCCESS) {
        fprintf(stderr, "could not connect to provider\n");
        exit(1);
    }
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Code Example 4-7 Listing of `monitorcalls.cc` *(5 of 5)*

```
/* Copyright (c) 1992 by Sun Microsystems, Inc. */
    while(1)
        d.dispatch();
}
```


PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Querying the XTEL Configuration Database 5

The XTEL database contains information on the available providers. This chapter explains how applications can query this database. For more information on the configuration database, see the *Solaris Teleservices 1.0 System Administrator's Guide*.

Overview

XTEL provider objects establish connections over telephone lines using lower-level protocols that control the telephone hardware and manage these connections. The entities that implement these low-level protocols are also called providers. The provider object must be able to connect to a valid provider in order to perform any telephone function. The provider configuration files maintained by the database are lists of keys and values that define specific provider characteristics necessary to the successful operation of each provider. The establishment of the XtelProvider object to provider connection is established when the Provider object is being constructed.

XTEL applications can use the XTEL database query functions to ensure that the XtelProvider object can successfully establish these connections. Applications use query functions to:

- Check that the database is appropriately set up
- Add new providers to the system's list of available providers.
- Access provider key and value information

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Using the Database Query Functions

The library of functions allowan application to query the database for a provider's key and value information. Each query should reference a specific provider alias and key. Table 5-1 shows the available database query functions.

Table 5-1 Database Query Functions (from *xtedb.h*)

Function	Description
extern int XTELDbInit(void)	Checks the XTEL configuration database to ensure that it is valid.
extern int XtelProviderNames(XtelKVList& names)	Retrieves a list of all configured provider names in the XtelKVList. Within the XtelKVList, the key specifies the provider's secondary alias while the value element contains the primary alias of that provider. Values are of type XtelString. XtelProviderNames() returns a count of provider names retrieved, otherwise it returns -1 upon error.
int XtelProviderInfo(const XtelString& alias, XtelKVList& info)	Retrieves information about the provider specified by *alias* (primary or secondary) and returns the information in the XtelKVList. XtelProviderInfo() returns acount of key-value pairs that were successfully retrieved from the configuration database about the provider, otherwise it returns -1 upon error.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Using Data Streams 6

This chapter covers concepts concerning XTEL applications that use data streams. In particular, using data streams to carry and direct audio data is discussed.

Manipulating Data Streams

Manipulating data streams involves using three methods from the XtelCall object:

- `Error configureDataStream(XtelKVList& new_configuration)`

  This method configures the data stream using the key-value pairs contained in the `new_configuration` argument.

- ```
  virtual void configureDataStreamEvent(
          XtelKVList& current_configuration)
  ```

- `XtelKVList& getDataStreamConfiguration()`

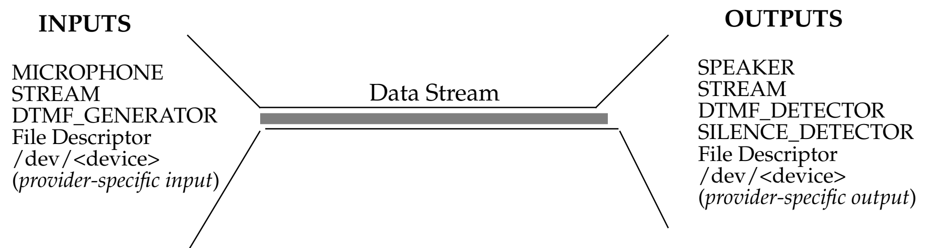
  This method retrieves the current data stream configuration.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT
  ```

Configuring Data Streams

To set up a data stream, you need specify its configuration in an XtelKVList. The key-value pairs in the list specify the input source and output destination that will be connected at the ends of a data stream; see Figure 6-1. Table 6-1 shows the possible input and output values.

Figure 6-1 Data Stream Inputs and Outputs



Changes to a data stream configuration are cached by the API and you can retrieve the current configuration using `getDataStreamConfiguration()`.

Table 6-1 Audio Inputs and Outputs for Key-Value Pairs

Key	Value	Description
INPUT	MICROPHONE	Accepts input from the system's default microphone device.
	STREAM	Accepts input from a standard UNIX stream.
	<file descriptor>	Accepts input from an open file descriptor.
	/dev/<device>	Accepts input from a UNIX device.
	DTMF_GENERATOR	This is a type of filter that generates DTMF tones.
	(*provider-specific input*)	Accepts input from the provider's local (on-board) audio device. The value for the device can be found in the provider configuration file. By convention, this value is enclosed by parenthesis.
OUTPUT	SPEAKER	Directs output to the system's default speaker device.
	STREAM	Directs output to a standard UNIX stream.
	<file descriptor>	Directs output to an open file descriptor.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Table 6-1 Audio Inputs and Outputs for Key-Value Pairs

Key	Value	Description
	/dev/<device>	Directs output to a UNIX device.
	SILENCE_DETECTOR DTMF_DETECTOR	
	(*provider-specific output*)	Directs output to the provider's local (on-board) audio device. The value for the device can be found in the provider configuration file. By convention, this value is enclosed by parenthesis.

Using Provider Specific Audio

A special convention is used to signify that an input or output value is provider specific. The convention is to use parentheses; for example, the 5E5 provider supports `(MICROPHONE)` and `(SPEAKER)` to connect data streams directly to the speakerbox.

Applications that use provider-specific inputs and outputs will not work on all providers. In general, these values should only be used in the provider configuration file.

Using System Default Audio

The `MICROPHONE` and `SPEAKER` value represent system defaults. When these values are used, the API looks up the default values in the XTEL configuration database and passes them to the provider.

Using STREAMs

You can specify the value STREAM for input or output to request that an opened UNIX stream be returned using `configureDataStreamEvent()`. The opened file descriptor will be returned with the key `INPUT_STREAM_FD` and `OUTPUT_STREAM_FD`, respectively.

Using STREAMs allows for some flexible configurations. For example, to request that an existing open file descriptor be spliced to a call's data stream, pass the file descriptor as an INPUT or OUTPUT value to `configureDataStream()`; `configureDataStreamEvent()` then returns the new valid configuration, else `error`() is called.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

You can also access a call's data directly by requesting an open file descriptor to a call's STREAM. This is done by passing the STREAM value for both INPUT and OUTPUT keys.

`Using` File Descriptors

You can specify ordinary file descriptors to connect to either end of a data stream.

Using Devices

You can specify a full device path name to connect to either end of a data stream.

Using Filters

There are three predefined filters that you can use:

- DTMF_GENERATOR
- DTMF_DETECTOR
- SILENCE_DETECTOR

Configuration Examples

A data stream configuration is specified in a XtelKVList as a list of input and output key-value pairs. When you submit the requested configuration through configureDataStream(), the provider either accepts the request, and confirms by sending back an XtelKVList that matches the request, or it denies the request and returns an error.

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. Table 6-2 shows several common data stream configurations.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Note – Any file descriptors returned from previous (`STREAM`) `configureDataStreamEvent()` calls is closed when a configuration is changed.

Table 6-2 Common Data Stream Configurations

Key	Value	Configuration
Empty	Empty	Null configuration
INPUT OUTPUT	MICROPHONE SPEAKER	Connect to system default audio devices
INPUT OUTPUT	(MICROPHONE) (SPEAKER)	Connect to provider-specific audio devices
INPUT OUTPUT	/dev/audio /dev/audio	Connect to /dev/audio device.
INPUT OUTPUT	STREAM STREAM	Get an open STREAM to a call's data.
OUTPUT	STREAM	Get a read-only STREAM to a call's data.
OUTPUT	DTMF_DETECTOR	Activate DTMF detection.
DTMF_DETECTOR_ <parameters>	<values>	Activate DTMF detection with optional parameters.
INPUT	DTMF_GENERATOR	Activate DTMF generation
OUTPUT	SILENCE_DETECTOR	Activate silence detection.
DTMF_DETECTOR_ <parameters>	<values>	Activate silence detection with optional parameters.
SILENCE_DETECTOR_ THRESHOLD	<milliseconds>	Detect silence with a duration of specified <milliseconds>.
INPUT	ZERO_LEN_BUF_ DETECTOR	Detect zero length buffers, which can be treated as audio markers.

Note – A provider may asynchronously change a data stream configuration. This can happen when calls are put on hold or disconnected. The provider notifies the application using `XtelCall::event()` with either `ACTIVE_CALL_EVENT` or `HELD_CALL_EVENT` events. In either case, these

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

events are equivalent to a configureDataStreamEvent() with an empty XtelKVList argument. in other words, before the provider dispatches either event, the data stream is set to a null configuration.

Using Voice Services

Voice services must be configured with a data stream so that a provider can more easily and intelligently decide how to optimize the implementation of a requested configuration. Possible voice service configuration values include `DTMF_DETECTOR`, `DTMF_GENERATOR`, and `SILENCE_DETECTOR`.

When these values appear in a data stream configuration, the provider performs certain initialization steps to provide the voice service. Voice service events are passed through the following notification methods:

- dtmfEvent()
- silenceEvent()
- nonSilenceEvent()
- zeroLengthBufferEvent()

DTMF generation requires additional command methods for a more interactive interface. If `DTMF_GENERATOR` is not configured, the command methods `dtmfGenerateTone()` and `dtmfGenerateString()` will return an error.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Error Codes

When an XTEL library call fails. the `error()` method of an object returns a code (an enumerated type). Table A-1 shows the possible errors and their meaning.

Table A-1 XTEL Error Codes

Error Code	Description
UNKNOWN_PROVIDER_ERROR	The provider received an error it could not translate.
INVALID_PROVIDER_ERROR	You sent a message to a provider that does not exist.
PROVIDER_INTERNAL_ERROR_ERROR	An internal error appeared within the provider. Report this error to the vendor who supplied the provider.
PERMISSION_DENIED_ERROR	You do not have ownership or permmissions to complete the given request.
INVALID_PARAMETER_ERROR	The given request had a bad parameter.
INVALID_REMOTE_NUMBER_ERROR	The outgoing number does not exist.
INVALID_OBJECT_ERROR	You sent a message to an object that does not exist.
INVALID_DATA_STREAM_ERROR	Either a data stream was not available (that is, it doesn't exist or the call is not in the ACTIVE_CALL state) or a `configureDataStream()` request could not be fullfilled as specified.
PROTOCOL_VIOLATION_ERROR	You sent a message that was not acceptable given the current state of an object.
XTEL_INTERNAL_ERROR	You sent a message that was not supported by the intended object.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT

Error Code	Description
INCOMING_CALL_REJECTED_ERROR	
CALL_REJECTED_ERROR	
RESOURCE_NOT_AVAILABLE_ERROR	A resource required to complete the request was not available.
DATA_CHANNEL_UNAVAILABLE_ERROR	A data channel was required to complete a request, but was not available.
SERVICE_NOT_IMPLEMENTED_ERROR	The provider you are using does not currently suport the given request.
SERVICE_NOT_SUBSCRIBED_ERROR	The provider or provider line is not configured for the requested service.
SERVICE_NOT_AVAILABLE_ERROR	The provider you are using cannot support the given request.
NETWORK_NOT_RESPONDING_ERROR	The service the provider is using to complete a request is not responding; the switch may have a bad name.
TIMER_EXPIRY_ERROR	A timeout occurred during the request.
NUMBER_CHANGED_ERROR	The outgoing number you called has been changed.
PROVIDER_SPECIFIC_ERROR	A proprietary provider-specific error occurred. See the provider's documentation and examine the KVList contents.
PROVIDER_INTERNAL_ERROR	
UNKNOWN_ERROR	The provider received an error it could not translate.

PROPERTY OF SUNSOFT, INC. — PRELIMINARY DRAFT