chapter

10 *Programming Notes*XE "Programming Notes"§XE "Operating system specifics"§

This chapter contains information about using the TSAPI libraries on various operating system platforms.

TSAPI on MacintoshXE "Operating system specifics:Macintosh"§

Macintosh Programming Overview

Read this section for information on developing TSAPI applications on Macintosh. You need not be familiar with the CSTA call model or API before reading further, but you should read Chapter 4, ***ACS Control Services***.

Macintosh Development Platforms

Telephony Services applications may be created in any Macintosh development environment; the TSAPI headers in this SDK contain explicit support for Metrowerks C and C++, Apple's MPW C and PPCC and Symantec C++ and THINK C.

Creating applications with other compilers may require you to modify these headers.

You should be aware of the following compiler environment considerations when using TSAPI:

- ◆ TSAPI requires enumerated types to be variable sizes

  Many Macintosh C compilers support two different storage classes for the enumerated type. All compilers supported by TSAPI allow enumerated types to be the size of the minimum integral type necessary to store their range of enumerated values. This option is sometimes called "packed enums". Many compilers also support forcing enumerated types to be the same size as int.

  Table X.1 describes compiler settings necessary to enable variable-sized enums.

Table 10-1<bookmark mac_enum>
Enum Settings in Macintosh Compilerstc "Enum Settings in Macintosh Compilers" \f t \l3§

Compiler	**Enum Packing Directive**
Apple MPW C 3.2	N/A
Apple PPCC 1.0	-enum min
Metrowerks Codewarrior C & C++, 68k	Language: Enums Always Int **unchecked**

Metrowerks Codewarrior C & C++, PPC	Language: Enums Always Int **unchecked**
Metrowerks mwcPPC	-enum off
Symantec THINK C 6.0, 7.0	Language Settings:enums are always ints **unchecked**
Symantec C++ for Macintosh 6.0, 7.0	Language Settings:enums are always ints **unchecked**

◆ TSAPI structures require mac68k alignment

TSAPI requires 68k Macintosh compilers to use two-byte structure alignment. For Macintosh compilers targeting PowerPC, TSAPI declares all structures using the #pragma options align=mac68k directive. If your PowerPC compiler does not support this pragma or does not define either powerc or __powerc, you must manually specify 2-byte alignment.

Table X.2 describes compiler settings necessary to enable 2-byte structure alignment:

Table 10-2<bookmark mac_struct>
Structure Alignment Settings in Macintosh Compilerstc "Structure Alignment Settings in Macintosh Compilers" \f t \l3§

Compiler	Structure Alignment Directive
Apple MPW C 3.2	N/A
Apple PPCC 1.0	N/A
Metrowerks Codewarrior C & C++, 68k	Processor: Struct Alignment: 68k
Metrowerks Codewarrior C & C++, PPC	N/A
Metrowerks mwcPPC	N/A
Symantec THINK C 6.0, 7.0	Processor Settings:align arrays of char **checked**
Symantec C++ for Macintosh 6.0, 7.0	Processor Settings:Struct Field Alignment:Align to 2 byte boundary

- TSAPI for 68k requires using an MPW .o link library
  Refer to your compiler documentation for instructions on

linking MPW .o format object modules with your 68k code.

TSAPI and Gestalt

Call Gestalt with gestaltTSAPICstaVersion as the *selector* parameter to find out the current version of the Macintosh Telephony Services client library.

If the library is available, the *response* parameter will point to the library version.

If the library is unavailable, Gestalt will return an error or the *response* parameter will contain zero.

For more information on using the Gestalt manager, see reference [6].

Dynamic Linking

This section describes how to dynamically link with the Telephony Services library on Macintosh.

680x0 Macintosh Dynamic Linking

On other platforms, client applications use inherent operating system facilities to dynamically link with the Telephony Services library. The Macintosh 68k runtime model does not provide such a facility. Instead, Apple provides several methods for achieving runtime linking — drivers, the Component Manager and the Apple Shared Library Manager.

The Macintosh Telephony Services library is an application that exports TSAPI using the Component Manager. The SDK provides a static link library in MPW .o object format that translates from TSAPI to the Component Manager calls necessary for using the Telephony Services library's CSTA component.

You need not use Gestalt to determine if the 68k Telephony Services library is running before using any TSAPI functions. If you do not use Gestalt, you should verify that you are running on a 68020 or better processor before making TSAPI calls.

Refer to your compiler documentation for instructions on linking MPW .o format object modules with your 68k code.

PowerPC Macintosh Dynamic Linking

Macintosh on PowerPC provides dynamic linking as described in reference [7].

You must use Gestalt as described above to determine if the PowerPC Telephony Services library is running before using any TSAPI functions. Failure to do so may result in crashing the host Macintosh.

Using Application Control Services

This section discusses how to use application control services (ACS) for such tasks as event notification and retrieval on Macintosh. If you are porting code that uses Telephony Services, you should read this section to get an overview of the differences between Macintosh and other platforms.

Event Notification

On Macintosh both acsEventNotify() and acsSetESR() are available and are most analogous to their Windows counterparts.

To use acsEventNotify(), you should understand how to receive and interpret Apple Events in your application. You can find information on using Apple Events in reference [4].

As with its counterparts on other platforms, acsEventNotify() can post a message to your application whenever any message is received from the Telephony Server (*notifyAll* = TRUE) or simply whenever a previously empty stream receives an event (*notifyAll* = FALSE).

The special feature of acsEventNotify() on Macintosh is that the process identifier is an AEAddressDesc. This allows your program to specify any legal address for an AppleEvent — network visible PPC entities included. The Telephony Services library will send notification AppleEvents using the kAENeverInteract flag; if the target application you specify is on a server to which the Macintosh is not authenticated, notification will fail. All notification events are sent with the kAENoReply flag.

Note:

If you are using acsEventNotify( ), you should use notifyAll = FALSE. Otherwise, the performance lag caused by processing Apple Events may unacceptably slow your application. The ability to post a message for every received event has been preserved for compatibility with TSAPI on other platforms.

To optimize your application for speed, you should use acsSetESR() to increment or set a variable in your application. Examine this variable to determine when to retrieve incoming events.

The following example demonstrates acsSetESR() being used to "preempt" an arbitrary lengthy processing task without polling for events — an important speed optimization. The example uses no global variables under 68k and hence may be easily implemented in any standalone code resource.

```
/*
 * Demonstration of using acsSetESR() to allow compute-bound
 * tasks to handle telephony traffic  w i t h o u t  polling
 * or global variables.
 */
```

```
#if defined (powerc) || defined (__powerc)
RoutineDescriptor myESRRD = BUILD_ROUTINE_DESCRIPTOR ( uppESRFuncProcInfo,
myESR );
#endif

/* ESR example */
pascal void myESR (unsigned long esrParam)
{
        /* esrParam points to the queuedEvents variable */
        unsigned short *queuedEventsPtr = *(unsigned short *)esrParam;

        /* increment global variable */
        *gQueuedEventsPtr++;
}

void computeWhileWatchingStream ( ACSHandle_t theStream )
{
        RetCode_t       rc;
        short   queuedEvents;   /* counting "semaphore" that    */
                                ............/* tracks number of events that */
                                ............/* have been received but not   */
                                ............/* processed */

        /* register callback, request notif for each event */
        #if defined (powerc) || defined (__powerc)
        rc = acsSetESR(theStream, &myESRRD, &queuedEvents, TRUE );
        #else
        rc = acsSetESR ( theStream, myESR, &queuedEvents, TRUE );
        #endif

        if ( rc != ACSPOSITIVE_ACK )
        {
                /* the callback could not be registered, so fail and                return
*/
                return;
        }

        /* begin iterative computation process */
        while ( SOME_LENGTHY_COMPUTATION_IN_PROGRESS )
        {
                if ( queuedEvents != 0 )
                {
                        /*
                         * Retrieve events here, or break out of loop,
         * etc.
                         */
                }

                /* process one step of a lengthy computation */
        }

        /* remove ESR before returning since it was using local     * storage to hold the queue
count
         */
        rc = acsSetESR ( theStream, NULL, TRUE );
}
```

Receiving Events

This section discusses event reception using acsGetEventPoll() and acsGetEventBlock() on Macintosh.

Blocking Versus Polling

Macintosh applications should generally use acsGetEventPoll() instead of acsGetEventBlock().

Whereas acsGetEventBlock() prevents most system activity from continuing until the calling application receives an event, acsGetEventPoll() returns control immediately if no event is waiting for the caller.

Calling acsGetEventPoll() frequently — particularly from 68k applications — can unduly consume processor time and resources. Instead of using polling as a method of determining whether messages are waiting in your application's receive queue, consider using event notification to trigger a polling call to receive events.

Neither acsGetEventBlock() nor acsGetEventBlock() may be called from a callback procedure registered with acsSetESR(). See the overview of event notification for an example of implementing a callback procedure to reduce polling.

Receiving Events From Any Stream

An application may specify a NULL stream handle when calling acsGetEventPoll() or acsGetEventBlock() to request that the Telephony Services library return the first event available on any of that application's streams.

When calling acsGetEventPoll() or acsGetEventBlock() from 68k code without a valid A5 world, applications must not use a NULL stream handle. Supplying a NULL ACS handle in this cases results in unpredictable system behavior; the library may return ACSERR_BADPARAMETER or simply crash.

Standalone code resources such as extensions, plug-ins and drivers do not have A5 worlds. 68k applications must make sure that their A5 world is valid before calling either get event function. See "Using TSAPI In Standalone Code" for more information about creating non-applications.

This restriction does not apply to PowerPC applications and standalone code. Any type of Macintosh code may use non-NULL stream handles.

TSAPI Resource Management

The Telephony Services library allocates session resources such as ACS stream memory and network connection resources from the heap active when acsOpenStream() is called.

Using TSAPI In Standalone Code

The Telephony Services library may be called from any type of code — applications, drivers, extensions, plug-ins et al. — but special rules apply when calling TSAPI from non-application code.

◆ Stream handles are not valid across processes

A stream handle opened while process A is running may not be used while process B is running. TSAPI calls made using a stream handle in the incorrect "process context" will return ACSERR_BADHDL.

◆ Streams will be closed when the owner process exits

A stream handle opened while process A is running will be closed when process A exits. You need to open streams using a persistent process if you are writing a code resource — an extension or Gestalt procedure, for example — that expects to remain alive while the host Macintosh is on.

TSAPI on OS/2XE "Operating system specifics:OS/2"§

TSAPI is fully supported under OS/2 2.1. Application developers can program to TSAPI to develop both Presentation Manager (PM) and non PM OS/2 applications. The IBM CSet++ 2.1, Borland C/C++ 1.5 and Watcom C/C++ OS/2 compilers are all supported. Using any other compiler may require user modification or conversion of header files.

Two TSAPI calls ***acsEnumServerNames***() and ***acsSetESR***() require the user to specify a callback function. These callback functions need to be defined with the _System calling convention.

OS/2 applications open ACS streams to the Telephony Server using the standard procedure outlined in this document. Once an ACS stream has been successfully opened, there are two ways for an OS/2 application to be notified that a TSAPI event is available to be retrieved. PM applications can use the ***acsEventNotify***() TSAPI call to designate a user defined message be posted to its application queue when a TSAPI event is available. PM and non-PM applications can use the ***acsSetESR***() TSAPI call to designate an Event Service Routine to be called whenever a TSAPI event is available. Alternatively, both PM and non-PM applications can forego event notification and receive events by creating a separate thread that uses ***acsGetEventBlock***() or ***acsGetEventPoll***() directly, to block until an event is received or to poll for events.

Both ***acsEventNotify***() and ***acsSetESR***() only signal the availability of a TSAPI message. The application must still call **acsGetEventBlock()** or **acsGetEventPoll( )** to actually retrieve the event from the Client API Library queue.

We now examine the notification mechanisms in more detail.

acsEventNotify()

```
#include <os2.h>
#include <csta.h>

#include <acs.h>

RetCode_t           acsEventNotify (
      ACSHandle_t   acsHandle,
      HWND                  hwnd,
      ULONG                 msg,
      Boolean               notifyAll);
```

acsHandle
is the value of the unique handle to the opened ACS

Stream for which event notification messages will be posted..

hwnd
is the handle of the window which is to receive event notification messages. If this parameter is NULL, event notification is disabled**.**

msg
is the user-defined message to be posted when an incoming event becomes available**.** The *mp1* and *mp2* parameters of the message will contain the following members of the ACSEventHeader_t structure:

mp1	acsHandle
SHORT2FROMMP(mp2)	eventClass
SHORT1FROMMP(mp2)	eventType

notifyAll

specifies whether a message will be posted for every event. If this parameter is **TRUE** then a message will be posted for every event**.** If it is **FALSE** then a message will only be posted each time the receive queue becomes non-empty, i.e. the queue count changes from zero (0) to one (1). This option may be used to reduce the overhead of notification, or the likelihood of overflowing the application's message queue.

Example

This example uses the **acsEventNotify** function to enable event notification.

```
#define WM_ACSEVENT WM_USER + 99

MRESULT EXPENTRY
WndProc (HWND hwnd, ULONG msg, MPARAM mp1, MPARAM mp2)
{
    // declare local variables...

    switch (msg)
    {
    case WM_CREATE:

        // post WM_ACSEVENT to this window
```

```
                // whenever an ACS event arrives

                acsEventNotify (acsHandle, hwnd, WM_ACSEVENT, TRUE);

                // other initialization, etc...
                return 0;

        case WM_ACSEVENT:

                // mp1 contains an ACSHandle_t
                // SHORT2FROMMP(mp2) contains an EventClass_t
                // SHORT1FROMMP(mp2) contains an EventType_t

                // dispatch the event to user-defined
                // handler function here

                return 0;

        // process other window messages...

        }
        return WinDefWindowProc (hwnd, msg, mp1, mp2);
}
```

acsSetESR()

The ESR mechanism can be used by the application to receive an asynchronous notification of the arrival of an incoming event from the Open ACS Stream. The application can use the ESR mechanism to trigger specific events (e.g. post an event semaphore). The ESR routine will receive one user-defined parameter. The ESR should **not** call ACS functions, otherwise the results will be indeterminate. The syntax of ***acsSetESR***() is as follows:

```
#include <os2.h>
#include <csta.h>
#include <acs.h>

typedef void (*EsrFunc)(ULONG esrParam)

RetCode_t acsSetESR (        ACSHandle_t        acsHandle,
                             EsrFunc            esr,
                                     ULONG              esrParam,
                                     Boolean            notifyAll);
```

acsHandle

is the value of the unique handle to the opened Stream for which this ESR routine will apply. Only one ESR is allowed per active acsHandle.

esr
points to the ESR (the address of a function). This function must use the _System calling convention. A multi-threaded application that registers the same ESR for multiple open streams needs to ensure that this function is reentrant. A NULL pointer is used to disable the current ESR mechanism.

esrParam
defines parameter which will be passed to the ESR when it is called**.**

notifyAll
specifies whether the ESR will be called for every event. If this parameter is **TRUE** then the ESR will be called for every event**.** If it is **FALSE** then the ESR will only be called each time the receive queue becomes non-empty, i.e. the queue count changes from zero (0) to one (1). This option may be used to reduce the overhead of notification.

TSAPI on UnixWareXE "Operating system specifics:UnixWare"§

Programming Overview

Read this section for information on developing TSAPI applications on UnixWare. You need not be familiar with the CSTA call model or API before reading further, but you should read Chapter 4, "ACS Control Services."

Development Platforms

Telephony Services applications must be built with an environment which supports the Executable and Linking Format (ELF) and dynamic linking. The TSAPI header files in this SDK are compatible with the C Optimized Compilation System provided with the UnixWare Software Development Kit. Using another compiler may require you to modify the header files, for example, to account for differences in structure alignment, size of enumerated data types, etc.

Linking to the TSAPI Library

The TSAPI for UnixWare is implemented as a shared object library, libcsta.so, and follows the standard conventions for library path search and dynamic linking. If libcsta.so is installed in one of the standard directories, it is only necessary to include "-lcsta -lnsl" in your link step, for example:

```
cc -o myprog myprog.c -lcsta -lnsl
```

Note that libcsta.so depends upon the Networking Support Library, libnsl.so.

Using Application Control Services

This section discusses how to use application control services (ACS) to retrieve events on UnixWare. If you are porting code that uses Telephony Services, you should read this section to get an overview of the differences between UnixWare and other platforms.

Event Notification

The acsEventNotify() and acsSetESR() functions are not provided on the UnixWare platform.

Unlike other Telephony Services platforms, UnixWare does not directly promote an event-driven programming model, but rather a file-oriented one. To work most effectively in the UnixWare environment, the TSAPI event stream should appear as a *file*. The acsGetFile() function returns the STREAMS file descriptor associated with an ACS stream handle. The returned value may be used like any other file descriptor in an I/O multiplexing call, such as poll() or select(), to determine the availability of TSAPI events. Alternatively, an application may register to receive the **SIGPOLL** signal using the I_SETSIG ioctl() call. Refer to *Programming with UNIX System Calls - STREAMS Polling and Multiplexing* in the UnixWare SDK documentation.

Do not perform other I/O or control operations directly on this file descriptor. Doing so may lead to unpredictable results from the TSAPI library.

Receiving Events

This section discusses event reception using acsGetEventPoll() and acsGetEventBlock() on UnixWare.

Blocking Versus Polling

acsGetEventBlock() suspends the calling application until it receives an event. If your application has no other work to perform in the meantime, this is the simplest and most efficient way to receive events from the TSAPI. Typically, however, an application needs to respond to input from the user or other sources, and cannot afford to wait exclusively for TSAPI events. acsGetEventPoll() returns control immediately if no event is available, allowing the application to query other input sources or events.

Calling acsGetEventPoll() repetitively can unduly consume processor time and resources, to the detriment of other applications. Instead of polling, consider multiplexing your input sources via the poll() system call, or installing a **SIGPOLL** handler.

Receiving Events From Any Stream

An application may specify a NULL stream handle when calling acsGetEventPoll() or acsGetEventBlock() to request that the Telephony Services library return the first event available on any of that application's streams.

Message Trace

To create a log file of TSAPI messages sent to and received from the Telephony Server, set the shell environment variable **CSTATRACE** to the pathname of the desired file, prior to starting your application. The log file will be created if necessary, or appended to if it already exists.

Sample Code

The following pseudo-code illustrates the use of the acsGetFile() function to set up an asynchronous event handler.

```
        int EventIsPending = 0;

/* handleEvent() called when SIGPOLL is received */

void
handleEvent (int sig)
{
        EventIsPending++;
        signal (SIGPOLL, handleEvent);          /* re-enable handler */
}

void
main (void)
{
        ACSHandle_t             acsHandle;
        int                     acs_fd;
                .
                .
                .

                /* install the signal handler */
        signal (SIGPOLL, handleEvent);

                /* open an ACS stream */
        acsOpenStream (&acsHandle, ...etc... );

                /* get its file descriptor */
        acs_fd = acsGetFile (acsHandle);

                /* enable SIGPOLL on normal "read" events */
        ioctl (acs_fd, I_SETSIG, S_RDNORM);

                /* proceed with application processing */
        while (notDone)
        {
                if (EventIsPending > 0)
                {
                                /* retrieve a TSAPI event */
                        acsGetEventPoll (acsHandle, ...etc...);
                }
                /* perform other background processing... */
        }
}
```