Altia® API Reference Manual

Altia Design for Windows®

HMI Software

August 2012

Table of Contents

0 Getting Started With Code Links

Altia Design and Runtime can be easily connected with external applications such as simulation models or compiled programs. Altia's Application Program Interface (API) allows software developers to establish communications between programs and Altia simulation graphics interfaces.

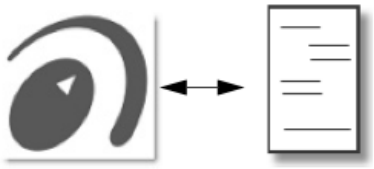
On Linux workstations, Altia supports TCP/IP sockets and domain sockets and uses these to communicate between your external code and Altia objects. On PC systems, Altia supports DDE's and TCP/IP socket configurations. To use the API with the external application, link the application using the appropriate Altia API library. The source code in the application can directly call the Altia API functions. In this document, we may refer to such an application as a client application, client program, or external program.

References to `/usr/altia`, `\usr\altia`, or `c:\usr\altia` in this document are meant to refer to the Altia Design software installation. If your Altia Design software is installed in a different directory, replace these references with your software installation directory to locate files and directories that are described in this document.

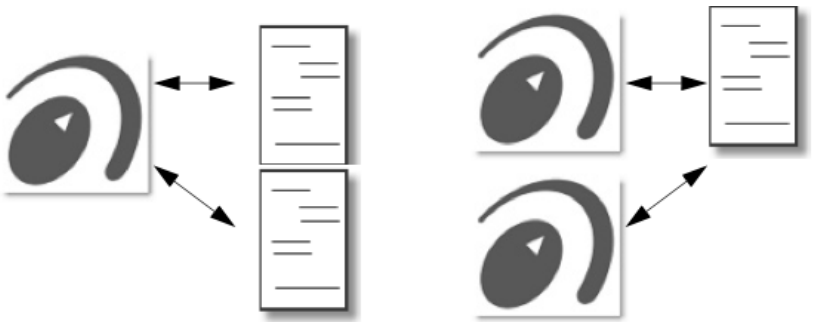
0.1 Architecture Possibilities

Because of the aforementioned inter-process communication schemes used by the API to pass data to and from applications, many useful Altia/Client arrangements are possible.

Altia can be set up to communicate with a single external program by sending and receiving animation variable events (as shown in the figure below).

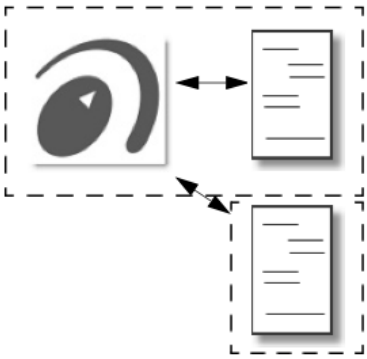


An Altia interface may just as easily communicate with more than one external program at a time as illustrated on the left side of the figure below. Likewise, a single client application may control and monitor multiple Altia sessions as illustrated on the right side of the figure below.



Altia may also be configured to communicate with programs running on different computers on your network. These computers may be running Linux, Windows, or a mix of the two. The figure below shows

Altia and a client application running on one computer with a second client application running on a different computer and connected to the same Altia session.



0.2 Opening and Closing Connections to Altia

The first step in linking an application to an Altia session is establishing a connection to an Altia interface.

0.2.1 Initiating a Connection

The **`AtOpenConnection()`** function (see *AtOpenConnection, AtCloseConnection*) attempts to establish a connection with an Altia session that is already running. Its parameters may be used to designate a specific socket or DDE that Altia may be serving. These parameters may also be left as `NULL` to link to the default Altia socket or DDE.

`AtOpenConnection()` returns either a valid `AtConnectId` value (which is used in other API functions) or a value of (`-1`), indicating failure to establish a connection with Altia.

Connections to multiple Altia sessions may be established and tracked by unique `AtConnectId` values returned by the **`AtOpenConnection()`** and **`AtStartInterface()`** functions. When sending or receiving data to an Altia session using various API functions, the `AtConnectId` parameter is used to specify which session/connection should receive/send the data.

The **`AtStartInterface()`** function (see *AtStartInterface, altiaStartInterface, AtStopInterface, altiaFetchArgcArgv*) can launch an Altia session, specify a design file to use, and establish a connection to that session. It is often used in conjunction with an **`AtOpenConnection()`** call. For example, if a connection to an already open Altia session cannot be established (because there isn't one open), you can open a new Altia session and connect to it.

0.2.2 Closing a Connection

Closing a connection to a specific session of Altia is done with **`AtCloseConnection()`** (see *AtOpenConnection, AtCloseConnection*). The `AtConnectId` parameter supplied to this function tells which specific connection to close.

0.2.3 A Simple Example of Opening and Closing a Connection

The following example does nothing useful. It simply opens and then closes a connection to a running Altia session.

```
#include "/usr/altia/libfloat/ms32/altia.h"
main()
{
        AtConnectId altiaId;
        altiaId = AtOpenConnection(NULL, NULL, 0, NULL);
        AtCloseConnection(altiaId)
}
```

Complete descriptions of the functions available from the Altia API are found in the ***Altia API Reference Manual*** or see **`/usr/altia/lib/altia.h`** for on-line descriptions. API source and reference files may be found in subdirectories of the **`lib`** or **`libfloat`** directories of your Altia software installation.

0.3 Sending Events to Altia

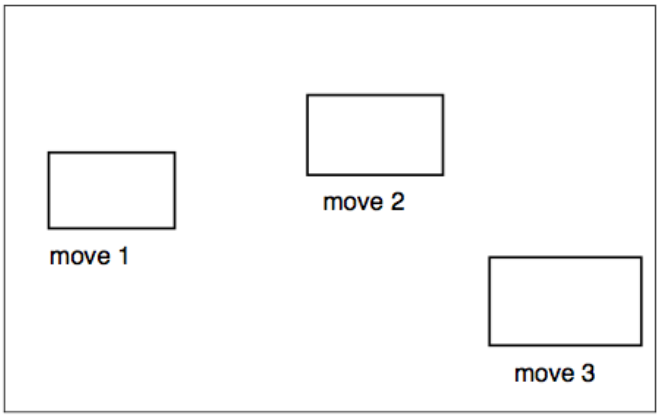
Outside programs send information to Altia in the form of animation variable events such as `box = 1`, `slide = 4.20`, or `label = "meter"`.

The **`AtSendEvent()`** function (see *AtSendEvent, altiaSendEvent*) is used to send numeric animation variable events to specific Altia animation names. It sends a special `AltiaEventType` variable which is an `int` if the integer libraries are used and a `double` if the floating point libraries are used. Note that this function will set the animation variable name to the value indicated in the **`AtSendEvent()`** function, even if the animation variable name was not previously used in Altia.

The **`AtSendText()`** function sends a string to an Altia text I/O object.

0.3.1 A Simple Example of Sending Events

Controlling an Altia animation function from an external program is as simple as calling any program function. The following is an example of code connection to animation. A box is drawn in Altia Design with three animation states defined. The animation function is called **`move`**.



1. In another window, create a file called `mover.c` with the following contents.

```
#include "/usr/altia/libfloat/ms32/altia.h"
main()
{
    AtConnectId altiaId;
    altiaId = AtOpenConnection(NULL, NULL, 0, NULL);
    AtSendEvent(altiaId, "move", (AltiaEventType) 2);
```

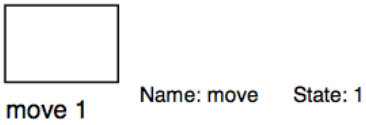
```
    AtCloseConnection(altiaId);
}
```

2. For Microsoft C/C++ users, compile and link by copying a sample `Makefile` from the **`\usr\altia\libfloat\ms32`** directory and replacing occurrences of `stress` with `mover`. Execute **`nmake`** (in the same directory) from an MS-DOS window.

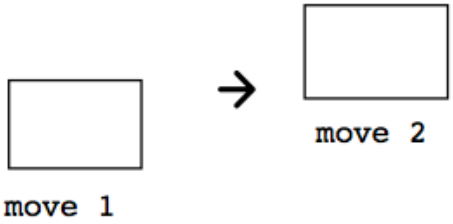
   On a Linux workstation, compile and link the file using:

   `gcc mover.c /usr/altia/libfloat/liblan.a -o mover`

3. In Altia Design, use the Animation Editor to place the `move` function in state `1`.



4. Start the program you created by typing `mover` in a separate window.



   The move program will move the box to defined state 2 in the Graphics Editor.

A simple program like this is all that is required to send data to an Altia Design animation. The program opens a connection to an already open Altia session, sends a value of two to the move animation, and then closes the Altia connection.

For more involved programming examples, please see the C source and make files in the demonstration directories:

`/usr/altia/demos/stressfloat`
`/usr/altia/demos/callback`

Complete descriptions of the functions available from the Altia Design API are found in the ***Altia API Reference Manual*** or see **`/usr/altia/lib/altia.h`** for on-line descriptions. API source and reference files may be found in subdirectories of the **`lib`** or **`libfloat`** directories of your Altia software installation.

0.4 Receiving Events from Altia

Altia's API provides three basic methods for receiving notice of an animation variable event (such as a button press or a slider's value change) from the Altia Interface file. Events may be received in a queue, Altia can be polled for values, or functions may be activated based on an Altia callback event.

0.4.1 Selecting Events to Receive

`AtSelectEvent()` (see *AtSelectEvent, altiaSelectEvent*) is used to receive notification of events for a specific animation variable. You can use multiple **`AtSelectEvent()`** function calls in your program to choose the specific animation variable events you wish to hear from Altia.

Using this approach, a queue of events and values is created for your program to parse and evaluate. You take the next event from the queue, evaluate its value, and make a programming decision based on it. The event `press == 1` from Altia might trigger the execution of a program function to send a `light = 1` event back to Altia.

The function **`AtNextEvent()`** (see *AtNextEvent, altiaNextEvent*) retrieves the next event from the queue of events, placing the name of the event animation variable and its value into its parameters. If no events are in the queue, **`AtNextEvent()`** will block execution of your program until events are available.

The function **`AtPending()`** (see *AtPending, altiaPending*) returns the number of events in the queue from Altia, if any.

Example program using Altia event selection:

```
#include "/usr/altia/libfloat/ms32/altia.h"
main()
{
    AtConnectId connectId;
    int i;
    AltiaEventType value;
    char *name;

    connectId = AtOpenConnection(NULL, NULL, 0, NULL);
    AtSelectEvent(connectId, "press");
    AtNextEvent(connectId, &name, &value);
    if (strcmp(name, "press") == 0 && value == 1)
    {
        for(i=0; i<=100; i++)
            AtSendEvent(connectId, "bar", (AltiaEventType) i);
    }
    else if (strcmp(name, "press") == 0 && value == 0)
    {
        for(i=100; i>=0; i--)
            AtSendEvent(connectId, "bar", (AltiaEventType) i);
    }
    AtCloseConnection(connectId);
}
```

The example above listens for a specific event from an already open Altia session and then, depending on its value, sends animations back to Altia. The **`AtSelectEvent()`** call tells the API that we are only interested in receiving the `press` event. When **`AtNextEvent()`** is called, the program will actually wait until it receives an event that has been selected via the **`AtSelectEvent()`** function (just `press` in this sample program). Once a `press` event happens in Altia, the name of the event is pointed to by `name` and the value is held by `value`. The `if` structure then sends 101 different values to an animation named `bar` (incrementing from 0 to 100 if the `value` was 1 or decrementing from 100 to 1 if `value` was 0).

0.4.1.1 Other Useful Selecting Functions

- `AtSelectAllEvents()` (see *AtSelectAllEvents, altiaSelectAllEvents*) tells Altia to route all animation variable events to the queue in your program.
- `AtUnselectEvent()` (see *AtUnselectEvent, altiaUnselectEvent*) can remove a specific animation variable from the list of items to be sent to the queue.
- `AtUnselectAllEvents()` (see *AtUnselectAllEvents, altiaUnselectAllEvents*) tells Altia to stop sending events from all animation variables to the queue in your program.

0.4.2 Polling Values

You can randomly check on the value of an animation variable in Altia by polling. This is often useful at a point in your program where you don't wish to parse through any events in the queue to find a specific animation variable's value.

`AtPollEvent()` (see *AtPollEvent, altiaPollEvent*) retrieves the value of an animation variable for the value parameter.

Example program using Altia polling:

```
include "/usr/altia/libfloat/ms32/altia.h"
main()
{
    AtConnectId connectId;
    int i;
    AltiaEventType valueOut;

    connectId = AtOpenConnection(NULL, NULL, NULL, NULL);
    do
    {
        AtPollEvent(connectId, "press", &valueOut);
    } while (valueOut != 1);
    for (i = 0; i <= 100; i++)
        AtSendEvent(connectId, "bar", (AltiaEventType) i);
    for (i = 100; i >= 0; i--)
        AtSendEvent(connectId, "bar", (AltiaEventType) i);
    AtCloseConnection(connectId);
}
```

The polling example above is very similar to the event selection example. It waits in a loop that repeatedly checks the value of the `press` animation in Altia. When `press` finally has a value of 1, it proceeds to send animation values from 0 to 100 back down to 0 to the `bar` animation in Altia.

Polling has the drawback that you can miss state changes if more than one state change occurs between polls. Selecting to receive specific events or using callbacks guarantees you will receive every state change in the exact order that they occurred.

0.4.3 Callback Functions

As an alternative to selecting events and parsing through a queue or polling, the Altia API provides functions for establishing callbacks.

The **`AtAddCallback()`** function (see *AtAddCallback, AtRemoveCallback, AtCallbackProc*) describes a specific function of your application to run based on a specific event happening in the Altia interface. You may, for

example, wish to execute a function based on a button being pressed. Multiple **`AtAddCallback()`** functions may be used to listen for all of the Altia events to be monitored as callbacks.

The **`AtMainLoop()`** function (see *AtMainLoop*) is used to monitor all listed callbacks and execute functions; this becomes the idle loop of your application.

Example program using Altia callbacks:

```
#include "/usr/altia/libfloat/ms32/altia.h"

void pressCallback();
void pressTimeout();

static AltiaEventType timerEnabled = 0;

main(int argc, char *argv[])
{
    AtConnectId connect = AtOpenConnection(NULL, NULL, argc, argv);
    AtAddCallback(connect, "press", pressCallback, NULL);
    AtMainLoop();
}

void pressCallback(AtConnectId connect, char* name, AltiaEventType value, AtPointer
data)
{
    if ((timerEnabled = value) == 1)
        pressTimeout((AtPointer) connect, 100L, NULL);
}

void pressTimeout(AtPointer data, unsigned long msecs, AtTimerId id)
{
    int i;
    if (timerEnabled)
    {
        for (i = 0; i <= 100; i++)
            AtSendEvent((AtConnectId) data, "bar", (AltiaEventType) i);
        for (i = 100; i >= 0; i--)
            AtSendEvent((AtConnectId) data, "bar", (AltiaEventType) i);
        AtAddTimer(msecs, pressTimeout, data);
    }
}
```

The example program above uses the API to execute a function (**`pressCallback()`**) whenever a `press` event occurs in Altia. The **`AtMainLoop()`** call is an idle process that waits until that happens. When `press` does happen, the **`pressCallback()`** function is triggered with the current value of `press`. If `value` is 1, then **`pressCallback()`** calls the function **`pressTimeout()`** which sends values to the `bar` animation from 0 to 100 back down to 0. The **`pressTimeout()`** function also sets itself up as a timer callback and continues to change the bar animation at 100 millisecond intervals until `press` is received with a value of 0 by **`pressCallback()`**.

0.4.3.1 Other Callback Functions

`AtAddTimer()` (see *AtAddTimer, AtRemoveTimer, AtTimerProc*) registers a callback to execute the callback function n milliseconds in the future. It does not, actually, depend on any events from the Altia interface.

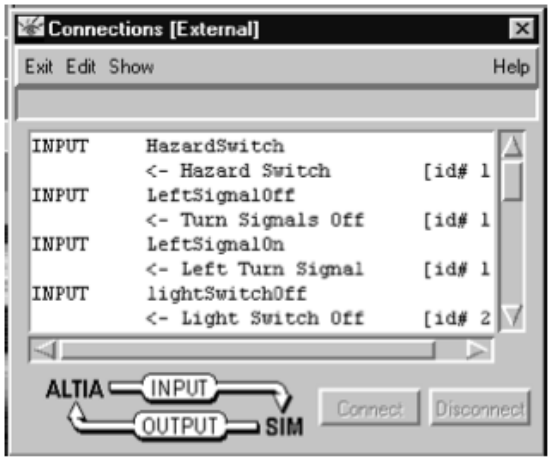
`AtAddTextCallback()` (see *AtAddTextCallback, AtRemoveTextCallback, AtTextProc*) registers a callback based on a string event happening to an Altia `textI/O` object.

`AtRemoveCallback()`, **`AtRemoveAllCallbacks`**(), **`AtRemoveTimer`**(), and **`AtRemoveTextCallBack`**() are used to remove previously registered callbacks from Altia's **`AtMainLoop`**().

Complete descriptions of the functions available from the Altia API are found in the ***Altia API Reference Manual*** or see **`/usr/altia/lib/altia.h`** for on-line descriptions. API source and reference files may be found in subdirectories of the **`lib`** or **`libfloat`** directories of your Altia software installation.

0.5 Using External Signals with Code

An external connection aliases external program events to Altia connectors. Programs built with the Altia API can take advantage of this mechanism so that specific animations of Altia objects do not have to be accessed.



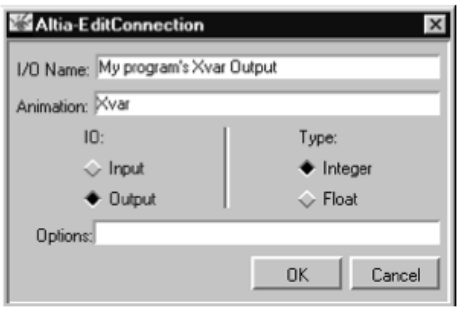
An External Connector can be used to map a program's output (**`AtSendEvent()`** calls) or input (**`AtNextEvent()`** calls) into `OUTPUT` and `INPUT` connectors, respectively. When these External Connectors exist, they behave just like connectors on Altia objects. The external program can be considered just another object with its own `INPUT`s and `OUTPUT`s, which are then linked with the connectors of Altia objects such as a slider or a meter.

0.5.1 Creating External Connections in the Connections[External] Dialog

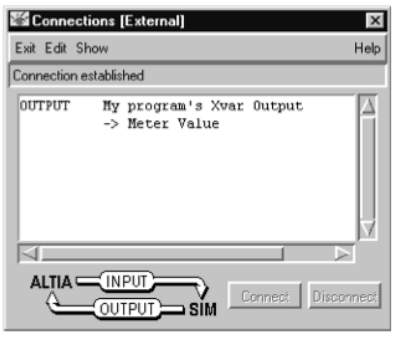
To open the Connections[External] Dialog, choose **External Signals** from the Altia Editor's **Connections** menu.

The **Add Connection...** option of the Connections[External] Dialog's **Edit** menu lets you add an external connection to your Altia Interface file. This option will let you map an external connector to the animation variable that your program references.

For example, suppose an external program will use the **`AtSendEvent()`** function to send values to an animation variable called `Xvar`. This output from the program can be mapped to an External Connector titled "`My Program's Xvar Output`."



Program events sent to `Xvar` will be routed along the "`My Program's Xvar Output`" `OUTPUT` connector to any connected Altia objects. In the case of the illustration below, they will go to the `Meter Value` connection. Connectors that serve as inputs to programs can be created in a similar fashion. The program could use **`AtSelectEvent()`** with **`AtNextEvent()`**, or a callback to receive the animation specified for the external `INPUT` connector.



0.6 Special API Functions

Aside from basic input and output, Altia's API provides sets of function calls for numerous special activities such as buffering and caching of data, manipulating sub-windows, dynamically cloning objects in your design file, and data logging.

0.6.1 Buffers and Caches of Data to Altia Design

On Linux, **`AtSendEvent()`** and **`AtSendText()`** function calls immediately send data to the Altia interface for animation and display. On Windows systems, however, the inter-process communication is buffered to enhance performance.

The **`AtCacheOutput()`** function (see *AtCacheOutput, AtFlushOutput, altiaCacheOutput, altiaFlushOutput*) toggles the API's mode of buffering data to be sent to Altia. Turning on buffering with **`AtCacheOutput()`** can significantly improve performance.

Buffered data is sent to Altia whenever the buffer is filled, when **`AtFlushOutput()`** is called, or when an API routine is called that requires immediate communication with the Altia interface (e.g., **`AtNextEvent()`** or

AtPollEvent()). If events are cached as a result of calls by a callback function, the cache is flushed when the callback function returns control to **AtMainLoop()**.

0.6.2 Opening, Closing, and Manipulating Views

Altia provides API functions to programmatically open and close sub-windows to your Altia interface design file, and to dynamically load additional Altia interface design files into existing sub-windows. These functions can be used to integrate user interface designs from separate development groups or to build a more cohesive presentation or software user interface.

These function calls will open windows using the operating system's window manager. These windows may be sized, minimized, closed, and otherwise controlled by the window's menus and controls.

0.6.2.1 Views

The **AtOpenView()** function (see *AtOpenView, altiaOpenView*) opens a new view with parameters for specifying initial size, orientation, and magnification. A `newViewId` integer parameter is used as a unique identifier to track and refer to the view. It is up to the programmer to supply and verify the uniqueness of this `newViewId`.

Functions such as **AtMoveView()**, **AtSizeView()**, and **AtMagnifyView()** can be used to modify a view once it has been opened. The specific window to be modified is determined from each function's `newViewId` parameter.

AtCloseView() (see *AtCloseView, altiaCloseView*) closes a specific, previously opened view.

0.6.2.2 Designs

AtOpenDesignFile() (see *AtOpenDesignFile, altiaOpenDesignFile*) opens a new interface design file into an existing Altia interface session. The programmer is responsible for providing a `designId` parameter value. If the `designId` parameter matches the `designId` of an Altia file already open, that original file is replaced with the new file. Otherwise, the currently active `designId` is set to the new `designId`.

The **AtSetDesignId()** function (see *AtSetDesignId, altiaSetDesignId*) designates which interface is currently available for API function calls (such as cloning functions) that require an object ID, as object IDs are only unique within a design file.

The **AtCloseDesignID()** function (see *AtCloseDesignId, altiaCloseDesignId*) will close a specific `designId`.

0.6.3 Using Clones

The Altia API provides functions for the program controlled creation of one or more copies of existing Altia interface objects. This cloning process results in new and independent objects that have all the Animation, Stimulus, Control, Properties, and Connections definitions and structure as the original object.

The **AtCreateClone()** function is used to create a clone of an existing object in the Altia interface file. The object to clone is referred to by its unique object ID; the cloned object is identified by a programmer supplied and tracked `cloneId` integer value. All clone object animation variables will be referred to by the

`cloneId:animationName`. Where the original object had an animation variable of `xLocation`, the cloned object will have an animation such as `5:xLocation`.

A full set of clone functions have been developed to mimic the input and output functions such as **`AtSendCloneEvent()`** and **`AtPollCloneEvent()`**.

The **`AtDeleteClone()`** and **`AtDeleteAllClones()`** functions can be used to programmatically remove clones. The **Delete All Clones** option from the Altia Graphics Editor's **Client** menu will delete all clones created or saved in your design file.

It is important to note that clones cannot be linked to external signals because clones are created dynamically. As a result, programs sending and receiving clone events must use the clone's animation names directly.

0.6.4 Data Logging Functions

The Altia Graphics Editor has a Record/Playback section that allows you to record and playback Altia interface use sessions. One can optionally log interface object inputs (button presses, etc.), toolbar commands (altia menus, dialogs and drawing tools), and client events (data from external applications) to a time-stamped ASCII text file.

`AtRecordMacro()` determines a macro file and prepares it for immediate recording.

`AtPlayMacro()` selects a macro file and plays it from the beginning.

Functions exist for starting, stopping, and appending to the recording process, as well as rewinding, playing, pausing, and stepping through a macro file. See `/usr/altia/lib/altia.h` for more information.

0.7 The Altia Makefile

C programs are created through a compiler which parses source files, checks syntax, links libraries together, and, if all goes smoothly, generates an executable program. The process of compilation itself may be controlled through a scripting environment known as a `Makefile` which derives its name from its task of telling the compiler how to make the program.

Altia provides `Makefile` files (located in `/usr/altia/libfloat`, `/usr/altia/lib`, or one of their subdirectories) for various compilers. We recommend taking one of these `Makefile` files and modifying it to refer to your application's compilation needs.

On Linux, the make command is `make`. On Windows with Microsoft C/C++, the make command is `nmake`. Open a command prompt window, change to the directory containing your makefile and C/C++ source files, and execute `make` or `nmake`.

0.8 The "altia" Subset of Functions

The initial release of Altia's API was engineered to work with only one Altia-Client connection at a time. These functions, which are a subset of the current Altia API, are designated with the `altia` prefix (e.g. **`altiaSendEvent()`**) and lack the `AtConnectId` parameter.

With the release of the "At" set of functions, Altia added multiple connection capability and additional behavior, including the callback set of function calls.

0.9 API Files

0.9.1 Altia API Libraries

Altia provides several libraries with a common application programming interface (API). Each Altia library uses a different transport medium, but the APIs are identical, so you only need to re-link your application with the appropriate library. The API itself is declared in the **altia.h** file in the same directory as the libraries.

Here are examples of the libraries provided with Altia Design:

Linux	**/usr/altia/lib/liblan.a** **/usr/altia/libfloat/liblan.a**
PC	**\usr\altia\lib\ms32\libdde.lib** **\usr\altia\lib\ms32\liblan.lib** **\usr\altia\lib\ms32\liblanMDoption.lib** **\usr\altia\lib\ms32\liblanMT.lib** **\usr\altia\libfloat\ms32\libdde.lib** **\usr\altia\libfloat\ms32\liblan.lib** **\usr\altia\libfloat\ms32\liblanMDoption.lib** **\usr\altia\libfloat\ms32\liblanMT.lib**

0.9.2 The Altia API Header (altia.h) File

This file should be obtained from the same directory as the selected library. It contains Altia API library function declarations for C and C++.

The following are sample C/C++ global declarations from this file.

```
int altiaSendEvent(const char *eventName, AltiaEventType eventValue);
int altiaSendText(const char *eventName, const char *text);
int altiaPollEvent(const char *eventName, AltiaEventType *eventValueOut);
int altiaNextEvent (char **eventNameOut, AltiaEventType *eventValueOut);
int altiaPending();
int altiaSelectEvent(const char *eventName);
int altiaSelectAllEvents();
int altiaUnselectEvent(const char *eventName);
int altiaUnselectAllEvents();
int altiaConnect(const char *serviceBaseName);
int altiaDisconnect();
int altiaRetryCount(int count);
```

The following is an example of a C++ function declaration, with comments, for altiaSendEvent.

```
/*
 * Initiate the transmission of an event to the Altia server.
 * The event's name is given by eventName and its value
 * is given by eventValue. If a communications failure
 * occurs, this function returns -1. Otherwise, 0 is
```

```
 * returned.
 */
int altiaSendEvent(const char *eventName, AltiaEventType eventValue);
```

0.10 Special API Notes for Linux Systems

0.10.1 Code Link via Domain Sockets

Probably the most common Altia code connection on Linux workstations is made through domain sockets. The following is an example of an `altiaSendEvent` call. The file was created in **`vi`** and saved as **`example.c`**:

```
#include "/usr/altia/libfloat/altia.h"
main ()
{
       altiaSendEvent("slider", (AltiaEventType) 5);
       altiaDisconnect();
}
```

then compiled as example (the Altia API library is also included on the command line):

```
cc -o example example.c /usr/altia/libfloat/liblan.a
```

The compiled program, **`example`**, looks for an Altia session by opening a particular domain socket (**`/usr/tmp/vSe.hostname`** in this case). If Altia is not running, the open will fail.
Altia keeps track of all code applications via the domain socket. You can easily run multiple client applications simultaneously, all of them connected to the same Altia session on the same socket.

0.10.2 Connecting Altia Interfaces to UNIX C/C++ Applications

0.10.2.1 Support for SunSoft Solaris 2.x on SPARC Systems

If you develop Solaris application programs that use the Altia Application Programming Interface (API), you must link your programs with the Solaris specific API library. With Altia Design for Sun Sparc, the Solaris library is:

`/usr/altia/libfloat/SOL/liblan.a`

With Altia Design for Solaris, the Solaris library is:

`/usr/altia/libfloat/liblan.a`

Finally, you will need to include two additional Solaris standard libraries in your link using the linker options **`-lsocket`** and **`-lnsl`**.

As an example, a program compilation and link command on Solaris with the Sun Sparc version of Altia Design might look like:

```
cc -o app.out app.c -I/usr/altia/libfloat/SOL /usr/altia/libfloat/SOL/liblan.a
-lsocket -lnsl
```

With a native Solaris version of Altia Design, the command would look like:

```
cc -o app.out app.c -I/usr/altia/libfloat /usr/altia/libfloat/liblan.a
-lsocket -lnsl
```

0.10.2.2 Available Methods for Executing Client Applications

Applications can connect to a design when it is loaded into Altia Design, Altia FacePlate, or Altia Runtime. An editor is started by executing:

```
/usr/altia[fp]/bin/altia[fp]
```

while the runtime product is started by executing

```
/usr/altia[fp]/bin/altia[fp] -run DESIGN_FILE
```

or

```
/usr/altia[fp]/bin/altiart.out -file DESIGN_FILE.
```

These programs can take two additional parameters of the form:

```
-exec PROGRAM
```

Typically, **PROGRAM** is an Altia client application - a C/C++ program linked with the Altia API library to control and respond to an Altia interface design. **PROGRAM** is executed as a separate process after the editor or runtime is fully unitized.

In addition to the **-exec PROGRAM** option, the editor also has two options under its **Client** menu that allow you to start a client application dynamically (**Start Client...**) or disconnect all currently running client applications (**Disconnect All Clients**). Under normal circumstances, a disconnect will terminate all client applications. These options are very useful while in the client application design phase. They allow you to test out code changes very quickly and terminate the program easily to perform additional code modifications.

0.11 Special API Notes for Windows Systems

0.11.1 Connecting Altia Interfaces to PC Windows Visual Basic Applications

Altia users can control their Altia interfaces from Basic programs using Microsoft Visual Basic. For more details, please consult the Application Note entitled ***Using Microsoft Visual Basic with Altia Design***.

0.11.1.1 Files of Note for Visual Basic Programmers

File	Description
altiaapi.txt **altia32.txt**	These files are Visual Basic source files declaring functions that can be used in 16 and 32 bit versions of Visual Basic respectively.
altiadde.dll **altdde32.dll** **altlan32.dll**	These DLL files support the functions used in Visual Basic programs.

0.11.2 Connecting Altia Interfaces to PC Windows C/C++ Applications

0.11.2.1 Files of Note for C/C++ Programmers

File	Description

altia.h	This header file contains declarations and some documentation for all API function calls and should be **#include** in your program source files.
libdde.lib	This file, included with the Windows version of Altia, resolves the functions declared in altia.h using DDE's as the communications mechanism between your application and Altia. It is statically linked by the compiler. A DDE library is the default library to use for Windows systems (unless for some reason you *must* use sockets). TCP/IP sockets (`liblan.lib`) are not supported by all Windows systems and, therefore, should be used with caution.
liblan.lib	This file, included with the Windows versions of Altia, resolves the functions declared in altia.h using a TCP/IP socket as the communications mechanism between your application and Altia. It is statically linked by the compiler.
makefile	This makefile is included in various forms. Developers may take this makefile and modify it for compiling their own projects.

0.11.2.2 Microsoft Compiler Support

Various 32-bit Microsoft compatible libraries and header files are located in **\usr\altia\lib** and **\usr\altia\libfloat** and their sub-directories. See the **\usr\altia\libfloat\readme.txt** file for more information. Using files from **\usr\altia\libfloat** is the preferred method for new development.

0.11.3 Types of Connections Available

Altia provides a C/C++ Application Programming Interface (API). It defines functions which allow C/C++ applications to control the animations of an Altia interface. Applications actually execute as separate processes from the Altia interface and they communicate with the interface using available inter-process communications facilities. The low-level details of inter-process communication are handled by the API object library being used.

The Linux version of the Altia API library uses TCP/IP sockets for inter-process communication. However, TCP/IP socket drivers ***are not standard*** on all PC systems. Therefore, an API library is provided that uses DDE's (Dynamic Data Exchange) for inter-process communication on PC's. C/C++ applications should link with the appropriate **libdde.lib** library located in **\usr\altia\libfloat** or **\usr\altia\lib**, or one of its sub-directories, to use DDE's. The C/C++ application programs that are part of the demonstration designs found in **\usr\altia\demos** are linked with a **libdde.lib** library.

If all of the machines for which you are developing support TCP/IP sockets and you wish to use sockets for inter-process communication, you can link with **\usr\altia\lib[float]\ms32\liblan.lib** for Microsoft 32-bit C/C++ development tools.

Complete descriptions of the functions available from the Altia API are found in the ***Altia API Reference Manual*** or see **`\usr\altia\lib[float]\altia.h`** for on-line descriptions.

0.11.4 Available Methods for Executing Client Applications

Applications can connect to a design when it is loaded into the Altia Design or Altia FacePlate Graphics Editor or Altia Runtime. The editor is started by executing **`\usr\altia[fp]\bin\altia[fp].exe`** while the runtime product is started by executing **`\usr\altia[fp]\bin\altiart.exe`**. These programs can take two additional parameters of the form:

```
-exec PROGRAM
```

Typically, **`PROGRAM`** is an Altia client application - a C/C++ program linked with **`libdde.lib or liblan.lib`** that uses the Altia API to control and respond to an Altia interface design. **`PROGRAM`** is executed as a separate process after the editor or runtime is fully initialized.

In addition to the **`-exec PROGRAM`** option, the editor also has two options under its **Client** menu that allow you to start a client application dynamically (**Start Client...**) or disconnect all currently running client applications (**Disconnect All Clients**). Under normal circumstance, a disconnect will terminate all client applications. These options are very useful while in the client application design phase. They allow you to test out code changes very quickly and terminate the program easily to perform additional code modifications.

0.11.5 Optimizations for the PC Versions of AtSendEvent() and altiaSendEvent()

The behavior of the Altia API functions **`AtSendEvent()`** and **`altiaSendEvent()`** are slightly different on Windows systems compared to the Linux versions. On Linux, these functions immediately send a message to the Altia interface. To improve inter-process communication performance on the PC, these functions buffer their messages within the application. When the buffer fills, all of the messages it contains are sent as a single data packet. The buffer is also emptied when an API function is called that requires a response from the Altia interface (**`AtSendEvent()`** and **`altiaSendEvent()`** do not require a response whereas a call like **`AtPollEvent()`** does). The buffer can also be explicitly emptied with a call to **`AtFlushOutput()`** or **`altiaFlushOutput()`**.

These subtle differences will normally go unnoticed. However, if you are expecting display changes from an **`AtSendEvent()`** or **`altiaSendEvent()`** call and they don't seem to be occurring, you may need to include an **`AtFlushOutput()`** or **`altiaFlushOutput()`** after the call. This is especially true if a program sends an event and then goes into some sleeping state (for example, using **`altiaSleep()`**). The event may not get sent before the sleep occurs. For more details, please refer to the **`AtSendEvent()`** description found in the ***Altia API Reference Manual.***

0.11.5.1 Proper Application Program Termination

Windows programs should execute: **`AtCloseConnections()`**, **`altiaDisconnect()`**, or **`AtStopInterface()`** before performing a program exit. Failure to do so may leave system resources allocated. This is not a problem for programs compiled under Windows XP/Vista/7 or for those using the Microsoft 32-bit C/C++ toolset.

0.11.6 Default WinMain for Windows

The **libdde.lib** library for 16-bit Windows provides a default **WinMain()** routine. Actual application code only needs to supply a simple **main** (**int argc, char *argv[]**) function as in the style of Linux programs and Windows 32-bit console programs. For this to work properly, the program must contain an include directive for the **\usr\altia\lib\altia.h** header file. If you wish to override the use of the default **WinMain** with your own **WinMain**, then link with **libddew.a** instead of **libdde.lib.**

1 Commands and Programs

1.1 Introduction

This section of the Altia Reference Manual describes commands and programs that are provided with the Altia installation. Commands and programs are typically executed from a shell prompt window running **sh**, **ksh**, or **csh** on Linux compatible systems, from the **Start** menu on Microsoft Windows or also from a Command Prompt Window on Microsoft Windows. Altia commands and programs are designed for execution within the X Window System environment on Linux, or on Microsoft Windows and they will not execute properly outside of these environments. Some users like to customize their Window Manager menus or Program Manager icons to include items for executing one or more Altia commands or programs. Documentation provided with your Windowing System should describe how this is done.

Altia commands and programs are installed in the Altia `bin` directory. If Altia is installed relative to `usr`, then the directory path would be `/usr/altia/bin` (or `\usr\altia\bin` for Windows users). To minimize typing, you might consider adding the directory path for `bin` to the `PATH` environment variable. On Linux systems, this is typically done by editing `.profile` or its equivalent in your home directory and updating the current `PATH` specification to include the new directory. On Windows, update the `PATH` environment variable from the System Properties Dialog (right click on the **My Computer** icon and choose **Properties**). If you are unfamiliar with the proper procedure for your environment, ask your Systems Administrator for help. Once the Altia `bin` directory is part of the `PATH` variable, Altia commands and programs can be executed from command prompt windows by simply typing their names rather than their entire path names.

Your Altia software installation includes a **license** directory which contains one or more license key files. These files contain licensing information that permits you to execute Altia programs on your system.

Commands are executable shell scripts (usually **sh** scripts on Linux and **batch** scripts on Window) while programs are binary files that are machine architecture specific. Commands usually serve as wrappers for programs. They make their best attempt to configure the environment to insure proper program execution. They may also try to determine the configuration of the display (e.g., number of colors available) and set references to application defaults files accordingly.

On Linux systems, **altia[fp]** is a **sh** script that executes altia[fp].out - the Altia Design or FacePlate Editor program. If the **-run** option is present, the script executes altiart.out to start an Altia Runtime session.

On Windows, typing **altia[fp]** starts the actual Altia Editor program. To start Altia Runtime, one must explicitly execute **altiart.exe** which can be done by simply typing **altiart**.

A Linux installation also has the scripts **overview, demo**, and **tutorial[n]** for starting automated demos and hands-on tutorials. On Windows, an Altia program group is provided which contains icons for starting demos, tutorials, and the Editor.

1.2 altia[fp]

1.2.1 Name

altia[fp]	On Linux, a script to start the Altia Editor or Altia Runtime program.
	On Windows, the Altia Editor program.

1.2.2 UNIX Synopsis

altia[fp]	[**-book**] [**-gray** | **-mono**] [**-loadfonts**] [**-colors** *count*] [**-title** '*titleName*']
	[**-display** *host:display*] [**-xrm** '*resourceSpec*'] [**-help**] [**-version**]
	[**-file** ***designFile*** | **-run** ***designFile*** [**-code** *runTimeCode*] [**-nowin**]]
	[**-defaults** *defaultsFile*] [**-exec** *command*]
	[**+/-log** [***logFile***] [**+/-loginput**] [**+/-logtool**] [**+/-logclient**]]
	[**-play** [***logFile***] [**-speed** *factor*] [**-nodelay**] [**-demo**] [**-nowarp**]]
	[**-port** ***pathName*** | **-port** *host:number* | **-port** *host:service*]
	[**-pipe**] [**-pipe** ***pathName***]

1.2.3 Windows Synopsis

altia[fp]	[**-book**] [**-gray** | **-mono**] [**-colors** *count*] [**-title** '*titleName*']
	[**-xrm** *resourceSpec*] [**-help**] [**-version**]
	[**-file** *designFile*]
	[**-defaults** *defaultsFile*] [**-exec** *command*]
	[**+/-log** [***logFile***] [**+/-loginput**] [**+/-logtool**] [**+/-logclient**]]
	[**-play** [*logFile*] [**-speed** *factor*] [**-nodelay**] [**-demo**] [**-nowarp**]]
	[**-lan**] [**-port** *host:number* | **-port** *host:service*]
	[**-dde** *serviceName*]

1.2.4 Description

altia[fp] is executed to start an Altia Design or FacePlate Editor session. On UNIX systems, it can also start an Altia Runtime session using the **-run** option description below. **altia[fp]** is located in the bin directory of the Altia software installation. If Altia is installed relative to usr, then the full path is /usr/altia[fp]/bin/altia[fp] on UNIX or \usr\altia[fp]\bin\altia[fp] on Windows systems.

On UNIX systems, **altia[fp]** can be executed from any shell prompt window that has local or network access to an X Windows session. On systems running Microsoft Windows, **altia[fp]** is executed from a program icon, from the Run dialog available through the Start menu, by double-clicking on the **altia[fp].exe** file when it is displayed in a file browser window, or by executing **altia[fp]** from a Command Prompt Window. And finally, if design files are saved with a .dsn extension and the Altia product has been properly installed, you can double-click on a design file from within a file browser window to automatically start the Altia Editor and open the design file.

To avoid specifying the full path for the command, users might consider adding the Altia bin directory to the PATH environment variable. See Section 1.1 for much more detail on setting environment variables for various types of systems.

On UNIX, **altia[fp]** is a **sh** script that customizes the environment and then executes **altia[fp].out** to start an Altia Editor session or **altiart.out** to start an Altia Runtime session if the **-run** option is specified.

On Windows systems, **altia[fp]** refers to the Altia Editor executable, **altia[fp].exe**. To start Altia Runtime, **altiart** (i.e., **altiart.exe**) must be explicitly executed.

The following options are recognized by **altia[fp]**:

altia[fp] Option	Description
-book	Customizes the Altia Editor for a notebook environment. The fonts, colors, window sizing and window locations are adjusted to take into account the lower resolution and color planes of a notebook. This option is ignored if the **-run** option is present.
-gray	Customizes the Altia Editor for a gray scale monitor. This option is ignored if the **-run** option is present. The **-mono** and **-gray** options are mutually exclusive - only one should be specified.
-mono	Customizes the Altia Editor for a monochrome monitor. This option is ignored if the **-run** option is present. The **-mono** and **-gray** options are mutually exclusive - only one should be specified.
-loadfonts	If the X display is local, **altia[fp]** automatically installs the Altia custom font directory paths into the X server's font path if they are not already installed. If the X display is remote, custom font path installation is suppressed; however, installation can be forced by including the **-loadfonts** option. This option is for X Window System users only.

altia[fp] Option	Description
-colors *count*	Customizes the number of colors in the Altia Editor's color palette. The default number of colors in the palette is chosen based on the number of display planes available. If the default is not satisfactory, it can be overridden with this option. This option is ignored if the **-run** option is present.
-title *'titleName'* **-title** *titleName*	Sets the title for the main window. Quotes are necessary if the title is more than one word or to avoid interpretation of special characters by the shell on UNIX
-display *host:display*	For UNIX users, displays the Altia interface on a X Window System display other than the X Window System display specified by the DISPLAY environment variable. The display name is given using the standard X syntax of host:display.
-xrm *'resourceSpec'* **-xrm** *resourceSpec*	Sets an application resource from the command line. The format of the specification is always in the style used by the X Window System (see examples below). This specification has precedence over all other specifications located in the X resource database, X defaults files, or an Altia application defaults file. The resource specification cannot contain tab or blank spaces. In addition, it should be enclosed in single quotes on UNIX systems to avoid interpretation of its contents by the command shell being used. As an example,**-xrm 'Altia*background:white'** could be used to set the background color of the Altia Editor interface to white. On Windows systems, use **-xrm Altia*background:white** (i.e., do not place resourceSpec within quotes).
-help	On UNIX systems, prints a complete argument summary to stderr. On Windows, the summary is written to the file stderr.log in the Altia bin directory. **altia[fp]** terminates after the summary listing.
-version	On UNIX systems, prints product version information to stderr. On Windows, the summary is written to the file stderr.log in the Altia bin directory. **altia[fp]** terminates after the version information is displayed.

altia[fp] Option	Description
-file *'designFile'* **-file** *designFile*	The Altia Editor is started and the design file specified by ***designFile*** is automatically loaded. If ***designFile*** is not an Altia design file, a dialog appears, but the Editor is left running. If ***designFile*** ends with a suffix such as .dsn, then a search is done for a file with the same base name and a suffix of .rtm. If the file is found, it's contents are used to configure the size and orientation of the Editor Views. If ***designFile*** has no suffix, then a search is done for ***designFile***.rtm instead. If the **-defaults** *defaultsFile* option is specified, then that file is used as the configuration file. A .rtm file contains application resource specifications and it is automatically created or recreated each time a design is saved from the Altia Editor. The **-file** ***designFile*** option and the **-run** ***designFile*** option are mutually exclusive - only one should be specified.
-run *'designFile'* **-run** *designFile*	On UNIX, starts Altia Runtime and automatically loads the design file specified by ***designFile***. If ***designFile*** is not an Altia design file, a dialog appears, and the runtime session is terminated after the dialog's OK button is pressed. If ***designFile*** ends with a suffix such as .dsn, then a search is done for a file with the same base name and a suffix of .rtm. If the file is found, it's contents are used to configure the size and orientation of the Runtime Views. If ***designFile*** has no suffix, then a search is done for ***designFile***.rtm instead. If the **-defaults** *defaultsFile* option is specified, then that file is used as the configuration file. A .rtm file contains application resource specifications and it is automatically created or recreated each time a design is saved from the Altia Editor. The **-file** ***designFile*** option and the **-run** ***designFile*** option are mutually exclusive - only one should be specified. On Windows systems, the **-run** option is not supported. **altiart** must be explicitly executed.
-code *runTimeCode*	If the **-run** ***designFile*** option is present, **-code** *runTimeCode* allows for the specification of a runtime license code word. For software release versions 2.0 or greater, a code word is not required to execute Altia Runtime. This option is only supported for backwards compatibility.

altia[fp] Option	Description
-nowin	If the **-run *designFile*** option is present, **-nowin** will start Altia Runtime without a Main View window. This allows an application program to control the displaying of Main Views using the view manipulation routines described in Section 3, Routines for Opening and Manipulating Views and Designs. This option is ignored if **-run *designFile*** is not present.
-defaults *defaultsFile*	Loads configuration information from the file *defaultsFile*. The format of the information is always in the style of X Window System resource specifications. If this option is present, then the implicit loading of a configuration file by the **-run** and **-file** options is suppressed. The resource specifications in this file take precedence over all other resource specifications accept those passed as command line arguments using the **-xrm** option.
-exec *command*	Executes *command* as an asynchronous sub-process. This can be used to start a client application program. The command is not executed until the design file specified with **-file** or **-run** is loaded and initialized. This guarantees a consistent start-up order for the application and the interface. The command can be a script or a program. If it takes parameters, then the complete command must be enclosed in double or single quotes so that it is handled as a single argument by the shell. As examples, **-exec 'echo hello'** or **-exec "echo $PATH"**.
+/-log [*logFile* **]**	Starts the logging of user actions performed within the Altia Editor or Altia Runtime. If ***logFile*** is not specified, logging goes to stdout. If **+log *logFile*** is specified, then logging is appended to ***logFile***. If **-log *logFile*** is specified, ***logFile*** is overwritten with the new log. By default, all keyboard and mouse activity is logged whether it is associated with editing or design execution.
+/-loginput	If **+/-log** is specified, then do/don't record design input events. The default is **+loginput** which will record design input events. This option is ignored if **+/-log** is not specified.
+/-logtool	If **+/-log** is specified, then do/don't record editor input events. The default is **+logtool** which will record editor input events. For Altia Runtime, this option only records interactions with information, warning, and error dialogs. This option is ignored if **+/-log** is not specified.

altia[fp] Option	Description
+/-logclient	If **+/-log** is specified, then do/don't record events sent to the interface from client applications. The default is **-logclient** which will not record events originating from client applications. This option is ignored if **+/-log** is not specified. It is unusual to find it necessary to log client events so avoid the **+logclient** option unless you know what you are doing.
-play [*logFile*]	Play back the log file ***logFile*** or read log commands from stdin if ***logFile*** is not specified. If ***logFile*** contains tool events and this is an Altia Runtime session, the tool events are ignored.
-speed *factor*	If **-play** is specified, perform the playback at *factor* speed where *factor* is a floating point number that gives the new speed relative to normal speed. A *factor* of 2 would double playback speed while .5 would slow it down by 50%.
-nodelay	If **-play** is specified, play back the logged events as fast as possible.
-demo	If **-play** is specified, **-demo** locks out new input events from the keyboard or mouse during playback. If this is an Altia Editor session and the log file contains tool events, **-demo** should be used. Otherwise, current mouse and keyboard events may disrupt the playback of editor input events. In addition, **-demo** forces warping of the display pointer during playback to give a good visual feel of what originally took place during logging.
-nowarp	If **-play** is specified, this option suppresses the warping of the display pointer during playback.
-lan	Specifies that an attempt should be made to use a socket for the connection between client application programs and the new Altia interface. This option is not necessary on Windows or UNIX.

altia[fp] Option	Description
-port *pathName*	On UNIX, specifies that a domain socket based off of ***pathName*** should serve as the connection between client application programs and the new Altia interface. For this to work, application programs must use ***pathName*** as the portName parameter to an altiaConnect or AtOpenConnection call and they must be linked with the liblan.a program library. On UNIX, Altia always opens a default domain socket even without a **-port** argument. The use of **-port** ***pathName*** is only necessary for specifying a domain socket other than the default. On Windows systems, domain sockets are not supported so use of the **-port** option in this form is ignored. The default socket type is a network socket with a socket number of 5100.
-port *:number*	Specifies that a network socket with the number *number* should serve as the connection between client application programs and the new Altia interface. For this to work, application programs must use *host:number* as the *portName* parameter to an altiaConnect or AtOpenConnection call where *host* is the name of the machine on which the Altia interface is running. If the application is running on the same machine, *host* may be omitted. Programs must also be linked with the liblan.a program library on UNIX or liblan.lib on Windows. On Windows, Altia always opens a network socket with the number 5100 as the default socket. The use of **-port** *:number* is only necessary for specifying a network socket number which is different than the default. On UNIX systems, socket numbers to 1000 or sometimes greater are usually reserved for system network facilities. To avoid conflicts, use a socket number of 2000 or greater (for example, use the PC default of 5100).

altia[fp] Option	Description
-port *:service*	Specifies that the network socket associated with the name *service* should serve as the connection between client application programs and the new Altia interface. *service* is an alias name for a network socket number. The /etc/services file or its equivalent must specify the mapping of *service* to an actual socket number. For this to work properly, application programs must use host:service as the *portName* parameter to an altiaConnect or AtOpenConnection call where host is the name of the machine on which the Altia interface is running. If the application program is running on the same machine, host may be omitted. Programs must also be linked with liblan.a on UNIX or liblan.lib on Windows. On UNIX systems, socket numbers to 1000 or sometimes greater are usually reserved for system network facilities. To avoid conflicts, only alias socket numbers of 2000 or greater (for example, use the PC default of 5100).
-dde *serviceName*	On Windows, specifies the use of a DDE Service Name other than the default DDE Service Name of AltDDE. For this to work, application programs must use *serviceName* as the *portName* parameter to an altiaConnect or AtOpenConnection call and they must be linked with the libdde.lib program library. On Windows, Altia always opens a default DDE Service even without a **-dde** option. Only use **-dde** to specify a DDE Service Name other than the default of AltDDE.
-pipe	On UNIX, specifies that pipes should be used as the connection between client application programs and the new Altia interface. In particular, a default pipe pair residing in /usr/tmp should be used. Sockets are the preferred method for client and Altia communications on UNIX. This option may be obsoleted in newer versions of Altia.
-pipe *pathName*	On UNIX, specifies that pipes should be used as the client/interface connection. In particular, a pipe pair based off of the file name ***pathName*** should be used. Sockets are the preferred method for client and Altia communications. This option may be obsoleted in newer versions of Altia.

1.2.5 UNIX Program and File References

File	Description
$ALTIAHOME/bin/altia[fp].out	Altia Editor program.
$ALTIAHOME/bin/altiart.out	Altia Runtime program.
$ALTIAHOME/bin/nplanes	Program executed by **altia[fp]** to determine display resolution.
xlsfonts	Standard X program executed by **altia[fp]** to test for installation of Altia custom fonts. PATH environment variable must include the path to xlsfonts (typically /usr/bin/X11 or /usr/openwin/bin).
xset	Standard X program executed by **altia[fp]** to install Altia custom font directory paths into the X server's font path. PATH environment variable must include the path to xset (typically /usr/bin/X11 or /usr/openwin/bin).
$ALTIAHOME/license/codewords	File containing license code word that allows execution of **altia[fp].out**.
$ALTIAHOME/app-defaults/...	Directory tree of Altia application defaults files for various display configurations (hi/lo res color, hi/lo res grayscale, monochrome, and SPARCbook)
$HOME/Altia	User's file containing application resource specifications that should override the default specifications from $ALTIAHOME/app-defaults/...
$XAPPLRESDIR/Altia	Alternate user's file containing application resource specifications that should override the default specifications from $ALTIAHOME/app-defaults/... Following X standards, if the environment variable XAPPLRESDIR is set, then $HOME/Altia is not searched.
/usr/tmp/vSe.hostname	Default domain socket for program connection.

1.2.6 Windows Program and File References

File	Description
%ALTIAHOME%\bin\altia[fp].exe	Altia Editor program. Typing simply **altia[fp]** actually executes **altia[fp].exe**.
%ALTIAHOME%\bin\fonts.ali	Fonts translation file read by **altia[fp].exe**.
%ALTIAHOME%\bin\colors.ali	Color name translation file read by **altia[fp].exe**.
%ALTIAHOME%\license\codeword.txt **%ALTIAHOME%\license*.lic**	Files containing license keys allowing execution of **altia[fp].exe**.
%ALTIAHOME%\defaults\...	Directory tree of Altia application defaults files for various display configurations (hi/lo res color, hi/lo res grayscale, monochrome, and notebook).
%HOMEPATH%\Altia WINDOWS\altia.ini	User's file containing application resource specifications that should override the default specifications from %ALTIAHOME%\defaults\... If %HOMEPATH% is not set, the WINDOWS directory is searched for altia.ini.

1.3 altiart.out (UNIX), altiart.exe (PC)

1.3.1 Name

File	Description
altiart.out	Altia Runtime program on UNIX installations.
altiart.exe	Altia Runtime program on Windows installations.

1.3.2 UNIX Synopsis

altiart.out	**-file** *designFile*
	[**-code** *ru**nT**imeCode*] [**-defaults** *defaultsFile*] [**-display** *host:display*]
	[**-exec** *command*] [**-title** '*titleName*'] [**-nowin**] [**-fullsize**] [**-help**] [**-version**]
	[**+/-log** [*logFile*] [**+/-loginput**] [**+/-logtool**] [**+/-logclient**]]
	[**-play** [*logFile*] [**-speed** *factor*] [**-nodelay**] [**-demo**] [**-nowarp**]]
	[**-port** *pathName* | **-port** *host:number* | **-port** *host:service*]
	[**-pipe**] [**-pipe** *pathName*]
	[**-xrm** '*resourceSpec*']

1.3.3 Windows Synopsis

altiart.exe	**-file** *designFile*
	[**-code** *runTimeCode*] [**-defaults** *defaultsFile*]
	[**-exec** *command*] [**-title** '*titleName*'] [**-nowin**] [**-fullsize**]
	[**+/-log** [*logFile*] [**+/-loginput**] [**+/-logtool**] [**+/-logclient**]]
	[**-play** [*logFile*] [**-speed** *factor*] [**-nodelay**] [**-demo**] [**-nowarp**]]
	[**-lan**] [**-port** *host:number* | **-port** *host:service*]
	[**-dde** *serviceName*]
	[**-xrm** *resourceSpec*]

1.3.4 Description

altiart.out is the actual Altia Runtime program on UNIX. On Windows, it is named **altiart.exe**. In cases where a design is being delivered to a site for strictly runtime execution, it is only necessary to install **altiart.out** or **altiart.exe** along with design files, .rtm configuration files, and application programs. The Altia Runtime program can be installed in any directory and executed from that directory. For example, to install the VCR demo for runtime-only execution, a tape or floppy might be created containing the following files:

For UNIX Systems:	For Windows 3.1 Systems:	For Windows9X or NT Systems:
altiart.out	altiart.exe	altiart.exe [go.bat]
vcr.dsn	vcr.dsn	vcr.dsn
vcr.rtm	vcr.rtm	vcr.rtm
vcr.out	vcr.exe	vcr.exe
fonts.ali *(if necessary)*	fonts.ali	fonts.ali
go *(optional)*	colors.ali	colors.ali

On UNIX, Windows 9x and Windows NT, The go file could be a script that executes **altiart.xxx**, loads the VCR design and configuration file, and also executes the vcr.xxx application program. The go script could be very simple - the single line:

```
altiart.out -file vcr.dsn -defaults vcr.rtm -exec vcr.out
```

would suffice on UNIX (on Windows 9X or Windows NT, replace occurrences of .out with .exe).

On Windows 3.1, the above command example cannot be executed from a script as scripting is purely a DOS feature. However, it could be used as the Command Line for a custom icon item.

The `fonts.ali` file is a copy of the `fonts.ali` file from the Altia bin directory on a Windows installation. It contains font name conversion specifications that may be needed by Altia Runtime. Some UNIX systems also have a `fonts.ali` file in the Altia bin directory. If this is the case on your system, then a copy of this file should accompany **altiart.out**.

The following options are recognized by **altiart.out** and **altiart.exe**:

-file *designFile*	Specifies the file containing the Altia design that should be loaded. The **-file** ***designFile*** argument must be specified. If it is not, the Altia Runtime session will terminate after displaying an error dialog. If ***designFile*** ends with a suffix such as .dsn, then a search is done for a file with the same base name and a suffix of .rtm. If the file is found, it's contents are used to configure the size and orientation of the runtime view. If ***designFile*** has no suffix, then a search is done for ***designFile***.rtm instead. If the **-defaults** *defaultsFile* option is specified, then that file is used as the configuration file. A .rtm file contains application resource specifications and it is automatically created or recreated each time a design is saved from the Altia Editor.
-code *runTimeCode*	Allows for the specification of a runtime license code word. This makes it possible to execute Altia Runtime without a code words file on the system.For software release versions 2.0 or greater, a code word is not required to execute Altia Runtime. This option is only supported for backwards compatibility.
-defaults *defaultsFile*	Loads configuration information from the file *defaultsFile*. The format of the information is always in the style of X Window System resource specifications. If this option is used, the normal search for a .rtm file is suppressed. The resource specifications in *defaultsFile* take precedence over all other resource specifications accept those passed as command line arguments using the **-xrm** option.
-display *host:display*	For UNIX users, displays the Altia interface on a X Window System display other than the X Window System display specified by the DISPLAY environment variable. The display name is given using the standard *X* syntax of *host:display*.

-exec *command*	Executes *command* as an asynchronous sub-process. This can be used to start a client application program. The command is not executed until the design file specified with the **-file** option is loaded and initialized. This guarantees a consistent start-up order for the application and the interface. *command* can be a script or a program. If it takes parameters, then the complete command must be enclosed in double or single quotes so that it is handled as a single argument by the shell. As examples, **-exec 'echo hello'** or **-exec "echo $PATH"**.
-title '*titleName*' **-title** *titleName*	Sets the title for the main window . Quotes are necessary if the title is more than one word or to avoid interpretation of special characters by the shell on UNIX systems.
-nowin	Starts Altia Runtime without a Runtime View window. This allows an application program to control the displaying of Runtime Views using the view manipulation routines described in Section 3.
-fullsize	Starts Altia Runtime with a main window that fills the entire display screen. No window banner will appear along the top of the main window. To close the session, press Ctrl-c. To ignore Ctrl-c, edit the .rtm and change the line: **Altia*quitControlChar: c** by removing the **c** character. In this case, the design should generate the event altiaQuit = 1 via stimulus, control, or an application program to close the Altia Runtime session gracefully.
-help	On UNIX systems, prints a complete argument summary to stderr. Altia Runtime terminates after the summary listing.
-version	On UNIX systems, prints product version information to stderr. Altia Runtime terminates after the version information is displayed.
+/-log [*logFile*]	Starts the logging of user actions performed during the Altia Runtime session. If ***logFile*** is not specified, logging goes to stdout. If **+log *logFile*** is specified, then logging is appended to ***logFile***. If **-log *logFile*** is specified, ***logFile*** is overwritten with the new log. By default,all input events are logged whether they are associated with the design or with Altia information, error, or warning dialogs.

+/-loginput	If **+/-log** is specified, then do/don't record design input events. The default is **+loginput** which will record design input events. This option is ignored if **+/-log** is not specified.
+/-logtool	If **+/-log** is specified, then do/don't record Altia information, error, or warning dialog interactions. The default is **+logtool** which will record these events. This option is ignored if **+/-log** is not specified.
+/-logclient	If **+/-log** is specified, then do/don't record events sent to the interface from client applications. The default is **-logclient** which will *not* record events originating from client applications. This option is ignored if **+/-log** is not specified. It is unusual to find it necessary to log client events so avoid the **+logclient** option unless you know what you are doing.
-play [*logFile*]	Play back the log file ***logFile*** or read log commands from stdin if ***logFile*** is not specified. If ***logFile*** contains editor tool events, they are ignored.
-speed *factor*	If **-play** is specified, perform the playback at *factor* speed where *factor* is a floating point number that gives the new speed relative to normal speed. A *factor* of 2 would double playback speed while .5 would slow it down by 50%.
-nodelay	If **-play** is specified, play back the logged events as fast as possible.
-demo	If **-play** is specified, **-demo** locks out new input events from the keyboard or mouse during playback. In addition, **-demo** forces warping of the display pointer during playback to give a good visual feel of what originally took place during logging.
-nowarp	If **-play** is specified, this option suppresses the warping of the display pointer during playback.
-lan	Specifies that an attempt should be made to use a socket for the connection between client application programs and the new Altia interface. Normally, only a DDE is opened on Windows systems. If no version of the **-port** option is found on the command line, then the default socket number of 5100 is used. This option is not necessary on Windows or UNIX.

-port *pathName*	On UNIX, specifies that a domain socket based off of ***pathName*** should serve as the connection between client application programs and the new Altia interface. For this to work, application programs must use ***pathName*** as the *portName* parameter to an altiaConnect or AtOpenConnection call and they must be linked with the liblan.a program library. On UNIX, Altia Runtime always opens a default domain socket even without a **-port** argument. The use of **-port** ***pathName*** is only necessary for specifying a domain socket other than the default. On Windows systems, domain sockets are not supported so use of the **-port** option in this form is ignored. The default socket type is a network socket with a socket number of 5100.
-port *:number*	Specifies that a network socket with the number *number* should serve as the connection between client application programs and the new Altia interface. For this to work, application programs must use *host:number* as the *portName* parameter to an altiaConnect or AtOpenConnection call where *host* is the name of the machine on which the Altia interface is running. If the application is running on the same machine, *host* may be omitted. Programs must also be linked with the liblan.a program library on UNIX or liblan.lib on Windows. On Windows, Altia Runtime always opens a network socket with the number 5100 as the default socket. The use of **-port** *:number* is only necessary for specifying a network socket number which is different than the default. On UNIX systems, socket numbers to 1000 or sometimes greater are usually reserved for system network facilities. To avoid conflicts, use a socket number of 2000 or greater (for example, use the PC default of 5100).

-port *:service*	Specifies that the network socket associated with the name *service* should serve as the connection between client application programs and the new Altia interface. *service* is an alias name for a network socket number. The /etc/services file or its equivalent must specify the mapping of *service* to an actual socket number. For this to work properly, application programs must use *host:service* as the *portName* parameter to an altiaConnect or AtOpenConnection call where *host* is the name of the machine on which the Altia interface is running. If the application program is running on the same machine, *host* may be omitted. Programs must also be linked with liblan.a on UNIX or liblan.lib on Windows. On UNIX systems, socket numbers to 1000 or sometimes greater are usually reserved for system network facilities. To avoid conflicts, only alias socket numbers of 2000 or greater (for example, use the PC default of 5100).
-dde *serviceName*	On Windows systems, specifies the use of a DDE Service Name other than the default DDE Service Name of AltDDE. For this to work, application programs must use *serviceName* as the *portName* parameter to an altiaConnect or AtOpenConnection call and they must be linked with the libdde.lib program library. On Windows systems, Altia Runtime always opens a default DDE Service even without a **-dde** option. Only use **-dde** to specify a DDE Service Name other than the default of AltDDE.
-pipe	For UNIX, specifies that pipes should be used as the connection between client application programs and the new Altia interface. In particular, a default pipe pair residing in /usr/tmp should be used. Sockets are the preferred method for client and Altia communications on UNIX. This option may be obsoleted for newer versions of Altia.
-pipe *pathName*	For UNIX, specifies that pipes should be used as the client/ interface connection. In particular, a pipe pair based off of the file name ***pathName*** should be used. Sockets are the preferred method for client and Altia communications on UNIX. This option may be obsoleted for newer versions of Altia.

-xrm '*resourceSpec*' **-xrm** *resourceSpec*	Sets an application resource from the command line. The format of the -xrm '*resourceSpec'* specification is always in the style used by the X Window System (see examples below). This specification has precedence over all other specifications located in the X resource database, X defaults files, or an Altia application defaults file. The resource specification cannot contain tab or blank spaces. In addition, it should be enclosed in single quotes on UNIX systems to avoid interpretation of its contents by the command shell being used. As an example: **-xrm 'Altia*AltiaView*width:200'** could be used to set the default width for Runtime Views to 200 pixels. On Windows systems, use **-xrm Altia*AltiaView*width:200** (i.e., do not place *resourceSpec* within quotes).

If the design loaded into Altia Runtime uses any Altia custom fonts, a runtime-only installation becomes a bit trickier. The custom fonts must be installed on the new machine. The approach used by the standard Altia installation for UNIX systems is to load the fonts into an Altia sub-directory and then use **xset fp+** *pathName* to install the name of the sub-directory into the *X* server font path. A different approach is to install the fonts into one of the standard fonts directories on the system. This might be /usr/lib/X11/fonts/misc or /usr/openwin/fonts, depending on the *X* server type. The Altia fonts directory, /usr/altia/fonts if Altia is installed relative to /usr, contains font files, information, and scripts for installing Altia custom fonts on UNIX. Please consult these files for more details.

1.3.5 UNIX Program and File References

File	Description
`fonts.ali`	Fonts translation file read by **altiart.out**. It should be a copy of the fonts.ali file found in the Altia bin directory. It must be placed in the same directory as **altiart.out**. If the Altia bin directory has no fonts.ali file, then it is not required for your type of UNIX system.
`$ALTIAHOME/license/codewords`	File containing license code word that allows execution of **altiart.out**. This file is only referenced if the **-code** option is not specified and no code word file specification (i.e., Altia*codeWordFile:*your_file_name*) is found in the .rtm file, *X* resource database, *X* defaults files, or $XAPPLRESDIR/Altia and HOME/Altia application defaults files. For software release versions 2.0 or greater, a code word is not required to execute Altia Runtime. This file is only supported for backwards compatibility.

File	Description
codewords	If present in the same directory as **altiart.out**, this file is checked for a valid code word if there is no **-code** option, no code word file specification in the .rtm file or $HOME/Altia and $XAPPLRESDIR/Altia files, and no $ALTIAHOME/license/codeword file is found. For software release versions 2.0 or greater, a code word is not required to execute Altia Runtime. This file is only supported for backwards compatibility.
$HOME/Altia	User's file containing custom application resource specifications if any are desired.
$XAPPLRESDIR/Altia	Alternate user's file containing application resource specifications. Following X standards, if the environment variable XAPPLRESDIR is set, then $HOME/Altia is not searched.
/usr/tmp/vSe.*hostname*	Default domain socket for application program connections.

1.3.6 Windows Program and File References

File	Description
fonts.ali	Fonts translation file read by **altiart.exe**. It should be a copy of the fonts.ali file found in the Altia bin directory. It must be placed in the same directory as **altiart.exe**.
colors.ali	Color translation file read by **altiart.exe**. It should be a copy of the colors.ali file found in the Altia bin directory. It must be placed in the same directory as **altiart.exe**.
%ALTIAHOME%\license\codeword.txt	File containing license code word that allows execution of **altiart.exe**. This file is only referenced if the **-code** option is not specified and no code word file specification (i.e., Altia*codeWordFile:*your_file_name*) is found in the .rtm file or %HOMEPATH%\Altia and WINDOWS\altia.ini files. For software release versions 2.0 or greater, a code word is not required to execute Altia Runtime. This file is only supported for backwards compatibility.

File	Description
codeword.txt	If present in the same directory as **altiart.exe**, this file is checked for a valid code word if there is no **-code** option, no code word file specification in the .rtm file or %HOMEPATH%\Altia and WINDOWS\altia.ini files, and no %ALTIAHOME%\license\codeword file is found. For software release versions 2.0 or greater, a code word is not required to execute Altia Runtime. This file is only supported for backwards compatibility.
%HOMEPATH%\Altia OR **WINDOWS\altia.ini**	User's file containing custom application resource specifications if any are desired. If %HOMEPATH% is not set, the WINDOWS directory is searched for altia.ini

1.4 soundconvert (UNIX)

1.4.1 Name

soundconvert	Universal sound sample translator available with Altia for UNIX.

1.4.2 UNIX Synopsis

soundconvert *infile outfile*

soundconvert *infile outfile* [*effect* [*effect options* ...]]

soundconvert *infile* **-e** *effect* [*effect options* ...]

soundconvert [*general options*] [*format options*] *ifile* [*format options*]
outfile [*effect* [*effect options* ...]]

1.4.3 General Options:

[**-V**] [**-v** *volume*]

1.4.4 Format Options:

[**-t** *filetype*] [**-r** *rate*] [**-s/-u/-U/-A**] [**-b/-w/-l/-f/-d/-D**] [**-c** *channels*] [**-x**]

1.4.5 Effects:

copy

rate

avg

stat

echo *delay volume* [*delay volume* ...]

vibro *speed* [*depth*]

lowp *center*

band [**-n**] *center* [*width*]

1.4.6 Description

soundconvert translates sound files from one format to another, possibly doing a sound effect.

soundconvert is a version of the publicly distributed SOX (SOund eXchange) program. It is provided by Altia w/o fee to Altia's UNIX customers. It is found in the altia/sound directory. Typically, it would be used to convert custom PC .WAV format audio files to the SUN .AU format for playing on SUN, HP, and SGI UNIX systems using Altia's sound playing capabilities.

A .WAV file might be copied to a UNIX system using binary ftp or a PC floppy (if your UNIX system has a PC compatible floppy drive). After copying, the .WAV file could be converted using a command such as:

```
soundconvert myfile.wav -U -b -r 8000 myfile.au rate
```

This requests a conversion of the .WAV file to a SUN U-law (**-U**), single byte (**-b**) format with a sample rate conversion to 8000 Hz (**-r 8000**) using the rate sound effect loop (**rate**). For *SGI* users, the **-U**, **-b**, **-r 8000**, and **rate** arguments are important or the *SGI* **sfplay** program will generate a warning message each time the sound file is played. For HP and SUN systems, it doesn't seem to be an issue.

THE SOUNDCONVERT PROGRAM IS PROVIDED BY ALTIA “AS IS” WITHOUT EXPRESS OR IMPLIED WARRANTY

1.4.7 Options

The option syntax can get tricky, but in essence:

```
soundconvert file.au file.voc
```

translates a sound sample in SUN Sparc .AU format into a SoundBlaster .VOC file, while

```
soundconvert -v 0.5 file.au -rate 12000 file.voc rate
```

does the same format translation but also lowers the amplitude by 1/2 and changes the sampling rate from 8000 hertz to 12000 hertz via the rate sound effect loop.

Soundconvert uses file suffices to determine the nature of a sound sample file. If it finds the suffix in its list, it uses the appropriate read or write handler to deal with that file. You may override the suffix by giving a different type via the **-t** type argument described below. Soundconvert has an auto-detect feature that attempts to figure out the nature of an unmarked sound sample.

1.4.8 File Type Options:

-t *filetype*	Gives the type of the sound sample file.
-r *rate*	Gives sample rate in Hertz of file.
-s/-u/-U/-A	The sample data is signed linear (2's complement), unsigned linear, U-law (logarithmic), or A-law (logarithmic). U-law and A-law are the U.S. and international standards for logarithmic telephone sound compression.

-b/-w/-l/-f/-d/-D	The sample data is in bytes, 16-bit words, 32-bit longwords, 32-bit floats, 64-bit double floats, or 80-bit IEEE floats. Floats and double floats are in native machine format.
-x	The sample data is in XINU format; that is, it comes from a machine with the opposite word order than yours and must be swapped according to the word-size given above. Only 16-bit and 32-bit integer data may be swapped. Machine-format floating-point data is not portable. IEEE floats are a fixed, portable format.
-c *channels*	The number of sound channels in the data file. This may be 1, 2, or 4; for mono, stereo, or quad sound data.

General options:

-e	Appearing after the input file, **-e** allows you to avoid giving an output file and just name an effect. This is only useful with the **stat** effect.
-v *volume*	Change amplitude (floating point); less than 1.0 decreases, greater than 1.0 increases. *Note*: we perceive volume logarithmically, not linearly. *Note*: see the **stat** effect.
-V	Print a description of processing phases. Useful for figuring out exactly how soundconvert is mangling your sound samples.

The input and output files may be standard input and output. This is specified by '-'. The **-t** *type* option must be given in this case, else `soundconvert` will not know the format of the given file. The **-t, -r, -s/-u/-U/-A, -b/-w/-l/-f/-d/-D** and **-x** options refer to the input data when given before the input file name. After, they refer to the output data.

If you don't give an output file name, `soundconvert` will just read the input file. This is useful for validating structured file formats; the **stat** effect may also be used via the **-e** option.

1.4.9 File Types:

Soundconvert needs to know the formats of the input and output files. File formats which have headers are checked, if that header doesn't seem right, the program exits with an appropriate message. Currently, the raw (no header), IRCAM Sound Files, Sound Blaster, SPARC .AU (w/header), Mac HCOM, PC/DOS .SOU, Sndtool, and Sounder, NeXT .SND, Windows 3.1 RIFF/WAV, Turtle Beach .SMP, and Apple/ SGI AIFF and 8SVX formats are supported.

`.aiff`	AIFF files used on Apple IIc/IIgs and SGI. Note: the AIFF format supports only one SSND chunk. It does not support multiple sound chunks, or the 8SVX musical instrument description format. AIFF files are multimedia archives and can have multiple audio and picture chunks. You may need a separate archiver to work with them.
`.au`	SUN Microsystems AU files. There are apparently many types of .au files; DEC has invented its own with a different magic number and word order. The .au handler can read these files but will not write them. Some .au files have valid AU headers and some do not. The latter are probably original SUN u-law 8000 hz samples. These can be dealt with using the .ul format (see below).
`.hcom`	Macintosh HCOM files. These are (apparently) Mac FSSD files with some variant of Huffman compression. The Macintosh has wacky file formats and this format handler apparently doesn't handle all the ones it should. Mac users will need your usual arsenal of file converters to deal with an HCOM file under UNIX or DOS.
`.raw`	Raw files (no header). The sample rate, size (byte, word, etc.), and style (signed, unsigned, etc.) of the sample file must be given. The number of channels defaults to 1.
`.ub, .sb, .uw, .sw, .ul`	These are several suffices which serve as a short-hand for raw files with a given size and style. Thus, ub, sb, uw, sw, and ul correspond to "unsigned byte", "signed byte", "unsigned word", "signed word", and "ulaw" (byte). The sample rate defaults to 8000 hz if not explicitly set, and the number of channels (as always) defaults to 1. There are lots of Sparc samples floating around in u-law format with no header and fixed at a sample rate of 8000 hz. (Certain sound management software cheerfully ignores the headers.) Similarly, most Mac sound files are in unsigned byte format with a sample rate of 11025 or 22050 hz.
`.sf`	IRCAM Sound Files. SoundFiles are used by academic music software such as the CSound package, and the MixView sound sample editor.

`.voc`	Sound Blaster VOC files. VOC files are multi-part and contain silence parts, looping, and different sample rates for different chunks. On input, the silence parts are filled out, loops are rejected, and sample data with a new sample rate is rejected. Silence with a different sample rate is generated appropriately. On output, silence is not detected, nor are impossible sample rates.
`.auto`	This is a *meta-type*: specifying this type for an input file triggers some code that tries to guess the real type by looking for magic words in the header. If the type can't be guessed, the program exits with an error message. The input must be a plain file, not a pipe. This type can't be used for output files.
`.smp`	Turtle Beach SampleVision files. SMP files are for use with the PC-DOS package SampleVision by Turtle Beach Softworks. This package is for communication to several MIDI samplers. All sample rates are supported by the package, although not all are supported by the samplers themselves. Currently loop points are ignored.
`.wav`	Windows 3.1 .WAV RIFF files. These appear to be very similar to IFF files, but not the same. They are the native sound file format of Windows 3.1.

1.4.10 Effects:

Only one effect from the palette may be applied to a sound sample. To do multiple effects you'll need to run **soundconvert** in a pipeline.

`copy`	Copy the input file to the output file. This is the default effect if both files have the same sampling rate, or the rates are *close*.
`rate`	Translate input sampling rate to output sampling rate via linear interpolation to the Least Common Multiple of the two sampling rates. This is the default effect if the two files have different sampling rates. This is fast but noisy.
`avg`	Mix 4- or 2-channel sound file into 2- or 1-channel file by averaging the samples for different speakers.

stat	Do a statistical check on the input file, and print results on the standard error file. **stat** may copy the file untouched, from input to output, if you select an output file. The "Volume Adjustment:" field in the statistics gives you the argument for the **-v** option which will make the sample as loud as possible.
echo [*delay volume ...*]	Add echoing to a sound sample. Each *delay/volume* pair gives the delay in seconds and the volume (relative to 1.0) of that echo. If the volumes add up to more than 1.0, the sound will melt down instead of fading away.
vibro *speed* [*depth*]	Add the world-famous Fender Vibro-Champ sound effect to a sound sample by using a sine wave as the volume knob. *Speed* gives the Hertz value of the wave. This must be under 30. *Depth* gives the amount the volume is cut into by the sine wave, ranging 0.0 to 1.0 and defaulting to 0.5.
lowp *center*	Apply a low-pass filter. The frequency response drops logarithmically with *center* frequency in the middle of the drop. The slope of the filter is quite gentle.
band [**-n**] *center* [*width*]	Apply a band-pass filter. The frequency response drops logarithmically around the *center* frequency. The *width* gives the slope of the drop. The frequencies at *center* + *width* and *center* - *width* will be half of their original amplitudes. Band defaults to a mode oriented to pitched signals, i.e. voice, singing, or instrumental music. The **-n** (for noise) option uses the alternate mode for un-pitched signals. Band introduces noise in the shape of the filter, i.e. peaking at the *center* frequency and settling around it.

Soundconvert enforces certain effects. If the two files have different sampling rates, the requested effect must be one of **copy** or **rate**. If the two files have different numbers of channels, the avg effect must be requested.

1.4.11 Bugs and Caveats

The syntax is horrific. It's very tempting to include a default system that allows an effect name as the program name and just pipes a sound sample from standard input to standard output, but the problem of inputting the sample rates makes this unworkable.

Soundconvert is intended as the Swiss Army knife of sound processing tools. It doesn't do anything very well, but sooner or later it comes in very handy.

Channel averaging doesn't work. The software architecture of stereo & quad is bogus.

1.4.12 Notices

Soundconvert is actually the publicly distributed SOX (SOund eXchange) program renamed for distribution with the Altia software without fee.

The echoplex effect is: Copyright (C) 1989 by Jef Poskanzer.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. This software is provided "as is" without express or implied warranty.

Creator & Maintainer:

Lance Norskog, thinman@netcom.com

Contributors:

Guido Van Rossum	guido@cwi.nl	AU, AIFF, AUTO, HCOM, reverse, many bug fixes
Jef Poskanzer	jef@well.sf.ca.us	original code for u-law and delay line
Bill Neisius	bill%solaria@hac2arpa.hac.com	
		DOS port, 8SVX, Sounder, Soundtool formats, Apollo fixes, stat with auto-picker
Rick Richardson	rick@digibd.com	WAV and SB driver handlers, fixes
David Champion	dgc3@midway.uchicago.edu	Amiga port
Pace Willisson	pace@blitz.com	Fixes for ESIX
Leigh Smith	leigh@psychok.dialix.oz.au	SMP and comment movement support.
David Sanderson	dws@ssec.wisc.edu	AIX3.1 fixes
Glenn Lewis	glewis@pcocd2.intel.com	AIFF chunking fixes
Brian Campbell	brianc@quantum.qnx.com	QNX port and 16-bit fixes
Chris Adams	gt8741@prism.gatech.edu	DOS port fixes
John Kohl	jtkohl@kolvir.elcr.ca.us	BSD386 port, VOC stereo support
Ken Kubo	ken@hmcvax.claremont.edu	VMS port, VOC stereo support
Frank Gadegast	<phade@cs.tu-berlin.de>	Microsoft C 7.0 port

2 General Library Routines

2.1 Introduction

This section describes Altia base library and toolkit routines that have applicability to a broad range of programming situations. In general, the routines allow application developer's to write programs that can communicate with one or more Altia interface sessions.

Altia base library routines are the core of the Altia library. Prior to release 1.2 of Altia Design, they were the only routines available. They form the low-level foundation for the toolkit routines which were added with release 1.2. Although the base library routines are sufficient for creating an application that can control or be controlled by an Altia interface, their low-level nature requires more code development on the part of the application programmer.

The toolkit routines extend the capabilities of the base library and they are the preferred approach for developing new applications.

To provide complete backwards compatibility, Altia continues to supply base library and toolkit versions of most routines. Some new routines, by their nature, only have a toolkit version.

Toolkit routines are differentiated with an At prefix while base library routines use an altia prefix. Where a routine has both a toolkit version and a base library version, they have been combined on a single manual page and the page is alphabetically sorted based on the toolkit name. For instance, the routines to send an animation event to an Altia interface are `AtSendEvent` and `altiaSendEvent`. Definitions for both appear on the manual page for `AtSendEvent`.

A program that uses general library routines needs to link with one of the object libraries supplied with Altia. If the product software is installed relative to /usr/altia on UNIX or \usr\altia on Windows as recommended, the floating point object libraries can be found in the /usr/altia/libfloat or \usr\altia\libfloat directories, respectively. By default, those writing code from scratch should use the floating point libraries. However, integer-based libraries are included (in the /usr/altia/lib directory on UNIX or \usr\altia\lib directory on Windows) for backwards compatibility with code written to connect to Altia Design 3.0 or earlier. Be sure to use the altia.h header file located in the same directory with your chosen library.

UNIX users typically link with the `liblan.a` object library which uses TCP/IP sockets for communicating between application programs and Altia Design. Windows users typically link with `libdde.lib` which uses a DDE for interprocess communication instead of a socket. If TCP/IP socket software and hardware exists on a Windows system, then programs can also link with liblan.lib. On all types of systems, the include file for the library is named altia.h and it resides in the same directory as the object libraries.

To get started, first take a look at `AtOpenConnection`. It is the toolkit routine used to establish a connection with an Altia interface. From that point, a program can send events to an Altia interface using `AtSendEvent` or `AtSendText`. The best approach for receiving events is to set up callback functions using `AtAddCallback` or `AtAddTextCallback`. The toolkit also supports time-out callback functions. These are set-up with the

`AtAddTimer` routine. Event and timer callback execution is controlled by the `AtMainLoop` routine. The Altia software installation provides a demos directory containing various complete Altia interface examples. Several of these show the toolkit routines in full action. The callback sub-directory is a good place to start and the vcr sub-directory contains a more sophisticated example.

If you are already familiar with the base library and prefer to program with it, first take a look at `altiaConnect` for establishing a connection with an Altia interface. A program can send events to an Altia interface using `altiaSendEvent` or `altiaSendText`. It receives events using `altiaNextEvent` or it can poll events using `altiaPollEvent`. The stress analyzer demo is written using base library routines and can serve as a programming example. It is found in the stress sub-directory of the Altia demos directory.

2.2 AtAddCallback, AtRemoveCallback, AtCallbackProc

2.2.1 Name

AtAddCallback	toolkit function to register a procedure as a callback for an Altia interface event.
AtRemoveCallback	toolkit function to remove a procedure from the callback list.
AtCallbackProc	prototype for Altia interface event callback procedure

2.2.2 C Synopsis

```
void AtAddCallback(AtConnectId connectId, const char *eventName,
                   AtCallbackProc callbackProc, AtPointer clientData);

void AtRemoveCallback(AtConnectId connectId, const char *eventName,
                      AtCallbackProc callbackProc, AtPointer clientData);

typedef void (*AtCallbackProc)(AtConnectId connectId, const char *eventName,
                               AltiaEventType eventValue, AtPointer clientData);
```

2.2.3 Description

AtAddCallback is a toolkit routine that adds a procedure to the Altia interface event callback list. When an Altia interface event is received from the connection specified by *connectId*, and the event's name matches the string pointed to by *eventName*, the callback procedure given by *callbackProc* will be invoked. AtAddCallback makes the appropriate call to AtSelectEvent to insure that the client application receives events of the type specified by *eventName*. Special data may be passed to the callback procedure through the *clientData* parameter, or it can be set to NULL. Callbacks can only be invoked if the application program executes AtMainLoop; or, if it has its own main loop, it must call AtDispatchEvent when an Altia interface event is received with AtNextEvent.

AtRemoveCallback is a toolkit routine that removes a procedure from the Altia interface event callback list if it is currently in the list. A procedure is only removed if it was added to the list with exactly the same attributes specified by *connectId*, *eventName*, *callbackProc*, and *clientData*. If there are no other callbacks in the list for the given event, AtUnselectEvent will be called to turn off receipt of events from the Altia interface to improve performance.

AtCallbackProc specifies the arguments and return value that a callback procedure must have in order for it to be compatible with the toolkit. The callback procedure will receive the id of the connection from which the event originated (*connectId*), the name of the event (*eventName*), the value for the event (*eventValue*), and the custom data that was passed to AtAddCallback when the callback was registered (*clientData*). Please note that the type for *eventValue*, AltiaEventType, is an int if the integer libraries are used and a double if the floating point libraries are used.

2.2.4 Return Values

None

2.2.5 Example

The following C main routine opens an Altia interface connection, adds a callback for the event press, then executes the toolkit main loop to wait for press events. When a press event is received by the callback function, it tests the value for the event to determine what action it should take. In this example, the custom data field for the callback is unused.

```
#include "altia.h"

void pressHandler();

main(argc, argv)
int argc;
char *argv[];
{
    AtConnectId connectId;
    if ((connectId = AtOpenConnection(NULL, NULL, argc, argv)) < 0)
    {
        fprintf(stderr, "AtOpenConnection failed\n");
        exit(1);
    }
    AtAddCallback(connectId, "press", pressHandler, NULL);
    AtMainLoop();
}

void pressHandler(connectId, eventName, eventValue, data)
AtConnectId connectId;
char *eventName;
AltiaEventType eventValue;
AtPointer data;
{
    if (eventValue == 1)
        printf("Received a press event\n");
    else
        printf("Received an unpress event\n");
}
```

2.2.6 See Also

AtMainLoop AtDispatchEvent

2.3 AtAddInput, AtRemoveInput, AtInputProc

2.3.1 Name

AtAddInput	registers a procedure as an I/O callback for an independent input source.
AtRemoveInput	removes an I/O callback procedure from the callback list.
AtInputProc	prototype for I/O callback procedure.

2.3.2 C Synopsis

```
AtInputId AtAddInput(int source, AtInputType inputType,
                     AtInputProc callbackProc, AtPointer clientData);

void AtRemoveInput(AtInputId inputId);

typedef void (*AtInputProc)(AtPointer clientData, int source, AtInputId inputId);
```

2.3.3 Description

AtAddInput is a toolkit routine that adds a procedure to the I/O event callback list. The *source* parameter specifies the origin of I/O. It is typically a file descriptor for an open pipe, socket, I/O card, or even stdin, stdout, or stderr. When I/O is pending on *source* and the condition specified by *inputType* is met, the procedure *callbackProc* is invoked. Special data can be passed to the callback procedure via the *clientData* parameter, or it can be set to NULL.

If *inputType* is AtInputRead, the procedure is executed when data is available for reading from *source*.

If *inputType* is AtInputWrite, the procedure is executed when data can be written to *source*.

If *inputType* is AtInputExcept, the procedure is executed when an exception occurs on *source*. In this case, *source* would reference some special device that supports the exception monitoring capabilities specified by the select(2) operating system call or its equivalent.

For all of the above *inputType* values, *source* must be a descriptor that can be monitored by the select(2) operating system call or its equivalent. AtMainLoop relies on select(2) to simultaneously monitor I/O sources, Altia interface sources, and time-outs. In addition, *source* must be a file descriptor with a value ranging from 0 through 31. AtMainLoop does not currently support descriptors larger than 31. As you might guess, I/O callbacks can only be invoked if the application program executes AtMainLoop.

AtAddInput returns a reference number for the newly registered I/O callback. To remove an I/O callback, the reference id must be passed as the *inputId* parameter to AtRemoveInput.

AtInputProc specifies the arguments and return value that an input callback procedure must have in order for it to be compatible with the toolkit. The callback procedure will receive the custom data that was passed to AtAddInput when the callback was registered (*clientData*), the source descriptor on which I/O is pending (*source*), and the input callback id that was assigned to the callback when it was registered (*inputId*).

2.3.4 Return Values

AtAddInput returns a non-negative id for the I/O callback just registered. This id must be passed to AtRemoveInput to remove the I/O callback. If AtAddInput fails (probably because of one or more invalid parameters), it returns -1.

AtRemoveInput has no return value.

2.3.5 Windows Dependencies

AtAddInput, AtRemoveInput, and input callback procedures are not supported.

2.3.6 Example

The following C main routine opens an Altia interface connection, adds an I/O callback to get data from **stdin**, and then executes the toolkit main loop to wait for **stdin** data.

```
#include <stdio.h>
#include "altia.h"

/* Must have forward declaration for stdinHandler
 * so main can reference it
 */
void stdinHandler();

main(argc, argv)
int argc;
char *argv[];
{
    AtConnectId connectId;
    if ((connectId = AtOpenConnection(NULL, NULL, argc, argv)) < 0)
    {
        fprintf(stderr, "AtOpenConnection failed\n");
        exit(1);
    }
    AtAddInput(0, AtInputRead, stdinHandler, NULL);
    AtMainLoop();
}

void stdinHandler(data, source, inputId)
AtPointer data;
int source;
AtInputId inputId;
{
    char input[256];
    fgets(input, sizeof(input), stdin);
    printf("Input callback read: %s", input);
}
```

The following example adds an I/O callback for a X11 socket connection.

```
main(argc, argv)
int argc;
char *argv[];
{
    Display *display;

    /* Normal initialization stuff would go here as in the previous
    * example plus X initialization.
    */
    AtAddInput(ConnectionNumber(display), AtInputRead, xInputHandler,
               AtPointer(display));
    AtMainLoop();
}

void xInputHandler(data, source, inputId)
{
    Display *display;
    XEvent event;
    int count;

    display = (Display *) data;
    /* This code will process all pending events. Could just
    * process one event for each call of the
    * callback, but this approach may be more efficient.
    */
    count = XPending(display);
    while (count--)
    {
        XNextEvent(display, &event);

        /* Do X event processing here or call
        * another function to do it.
        */
    }

    /* May want to flush X output queue before going back
    * to AtMainLoop()
    */
    XFlush(display);
}
```

2.3.7 See Also

AtMainLoop

2.4 AtAddTextCallback, AtRemoveTextCallback, AtTextProc

2.4.1 Name

AtAddTextCallback	registers a procedure as a text callback for a string of Altia interface events.
AtRemoveTextCallback	removes a text callback procedure from the callback list.
AtTextProc	prototype for Altia interface event text callback procedure.

2.4.2 C Synopsis

```
void AtAddTextCallback(AtConnectId connectId, const char *eventName,
                       AtCallbackProc callbackProc, AtPointer clientData);

void AtRemoveTextCallback(AtConnectId connectId, const char *eventName,
                          AtCallbackProc callbackProc, AtPointer clientData);

typedef void (*AtTextProc)(AtConnectId connectId,
                           const char *eventName, const char *string,
                           AtPointer clientData);
```

2.4.3 Description

AtAddTextCallback is a toolkit routine that adds a procedure to the Altia interface event callback list for receiving a string of events. Instead of invoking the associated callback for each individual event (which is what would happen if AtAddCallback were used), the toolkit casts the events into characters and buffers them internally until a null valued event (i.e., end of string identifier) is received. It then passes the entire set of events as a string to the procedure designated by *callbackProc*. The callback is added for events matching the name pointed to by *eventName* and received from the Altia interface connection designated by *connectId*. AtAddTextCallback makes the appropriate call to AtSelectEvent to insure that the client application receives events of the type specified by *eventName*. Special data may be passed to the callback procedure through the *clientData* parameter, or it can be set to NULL.

Callbacks can only be invoked if the application program executes AtMainLoop; or, if it has its own main loop, it must call AtDispatchEvent when an Altia interface event is received with AtNextEvent.

Text callbacks are designed specifically for receiving strings from Altia interface text input areas. Text input areas allow Altia interface designs to accept keyboard input from users. The input is displayed in a text field and dynamically updated as the user types in characters from the computer's keyboard.

Text input areas for software release 1.40 and earlier are constructed using text input objects. Text input objects have a built-in animation, referred to as the "getchar" animation, which allows applications to receive the characters stored in the object. Text input areas built using text input objects typically generate "getchar" events automatically when they detect a Return key press. This will trigger the execution of a text callback if one is registered for the "getchar" events.

For software release 2.0 and later, text input objects are replaced with a more versatile text i/o object. A text i/o object has a "text" animation that replaces the "getchar" animation of a text input object. Text input

areas built around the text i/o object generate "text" events automatically when they detect a Return key. They also generate a "done" event. This event can be received by a normal callback using AtAddCallback. The callback code can query the text i/o object for its text using the "text" animation in a call to AtGetText. Example text i/o objects are available from the Inputs, Textio and GUI Models Libraries. These are design files, **inputs.dsn, textio.dsn,** and **gui.dsn** or **guiwin.dsn** to be precise, found in the models sub-directory of the Altia software installation. They are organized for display within an Altia Models View (press the **Libraries** button in the Altia Graphics Editor to initiate the opening of a Models View). To learn specific details about text i/o objects, select the **Textio Models** option from the **Help** menu found in any Altia Models View.

For software release 2.0 and later, the older text input objects are still supported for backwards compatibility and they behave exactly as in previous releases. However, they are not available from the Models Libraries for copying into designs. They must be copied from an older design in which they were used.

AtRemoveTextCallback is a toolkit routine that removes a text callback procedure from the Altia interface event callback list if it is currently in the list. A procedure is only removed if it was added to the list with exactly the same attributes specified by *connectId*, *eventName*, *callbackProc*, and *clientData*. If there are no other callbacks in the list for the given event, AtUnselectEvent will be called to turn off receipt of events from the Altia interface to improve performance.

AtTextProc specifies the arguments and return value that a text callback procedure must have in order for it to be compatible with the toolkit. The callback procedure will receive the id of the connection from which the events originated (*connectId*), the name of the events (*eventName*), a null-terminated string containing the most recent event values (*string*), and the custom data that was passed to AtAddTextCallback when the callback was registered (*clientData*). The *string* parameter points to a character array that is managed by the toolkit. It is only valid during the call to the text callback. If the contents of the array need to be preserved, a copy of it should be made by the callback procedure.

2.4.4 Return Values

None

2.4.5 Example

The following C main routine opens an Altia interface connection, adds a text callback for "text" events, and then executes the toolkit main loop to wait for "text" events. The standard text input area sends "text" events when a user clicks the left mouse button over the input area, enters text, and presses Return or clicks the mouse button outside of the input area. Without modifications, the example code shown here would be able to receive character strings from a text input area. Text callbacks for release 1.40 or earlier text input areas are set up in the same fashion to receive "getchar" events instead of "text" events.

As discussed earlier, software release 2.0 provides a more versatile text i/o object. Input areas built with the text i/o object can also register a normal callback for a "done" event and then query the text i/o object's "text" animation for the input text using a call to AtGetText. See AtGetText, altiaGetText for a specific example.

```
/* This example demonstrates how to receive text input from text
 * input areas for software release 2.0 or later. For software
 * release 1.40 or earlier, the callback should be set up for
 * "getchar" events rather than "text" events.
 */
#include "altia.h"

/* Must have forward declaration for textHandler so main
 * can reference it
 */
void textHandler();

main(argc, argv)
int argc;
char *argv[];
{
    AtConnectId connectId;

    if ((connectId = AtOpenConnection(NULL, NULL, argc, argv)) < 0)
    {
        fprintf(stderr, "AtOpenConnection failed\n");
        exit(1);
    }

    AtAddTextCallback(connectId, "text", textHandler, NULL);
    AtMainLoop();
}

void textHandler(connectId, eventName, string, data)
AtConnectId connectId;
char *eventName;
char *string;
AtPointer data;
{
    printf("Received text \"%s\" from the text input object\n", string);
}
```

See Also

AtGetText, altiaGetText AtAddCallback, AtRemoveCallback, AtCallbackProc, AtMainLoop

2.5 AtAddTimer, AtRemoveTimer, AtTimerProc

2.5.1 Name

AtAddTimer	registers a procedure as a time-out callback.
AtRemoveTimer	removes a time-out callback procedure from the callback list.
AtTimerProc	prototype for time-out callback procedure.

2.5.2 C Synopsis

```
AtTimerId AtAddTimer(unsigned long msecInterval,
                     AtTimerProc callbackProc, AtPointer clientData);

void AtRemoveTimer(AtTimerId timerId);

typedef void (*AtTimerProc)(AtPointer clientData, unsigned long msecInterval,
                            AtTimerId timerId);
```

2.5.3 Description

AtAddTimer is a toolkit routine that adds a procedure to the timer event callback list. The procedure, specified by *callbackProc*, is executed after *msecInterval* milliseconds have elapsed. The callback can be sent special data by setting *clientData*, or just pass NULL. The callback is only invoked once. The callback can be reinstated into the callback list with an identical call to AtAddTimer from the callback itself, or from another procedure.

AtAddTimer returns an identification number for the callback just registered. It can be used to remove the callback from the timer event callback list by passing it as the *timerId* parameter to AtRemoveTimer. Since a callback is removed from the list when its time interval expires, it is only necessary to call AtRemoveTimer if there is a need to remove a timer event callback before its interval expires.

The toolkit makes its best attempt at calling the procedure within the specified interval, but accuracy is dependent on system load and the time expended by other procedures executed by the program. Also note that timer callbacks can only be invoked if the application calls AtMainLoop. It knows how to juggle the execution of Altia interface event callbacks, I/O callbacks, and timer callbacks.

AtTimerProc specifies the arguments and return value that a timer callback procedure must have in order for it to be compatible with the toolkit. The callback procedure will receive the custom data that was passed to AtAddTimer when the callback was registered (*clientData*), the original time-out interval in milliseconds (*msecInterval*), and the timer callback id that was assigned to the callback when it was registered (*timerId*).

2.5.4 Return Values

AtAddTimer returns a non-negative id for the timer callback just registered. This id must be passed to AtRemoveTimer to remove the timer callback before the time-out interval expires. If AtAddTimer fails (probably because of one or more invalid parameters), it returns -1.

AtRemoveTimer has no return value.

2.5.5 Very Important Windows 16-bit Dependencies

The "L" notation on a constant (for example, **1000L**) must be used when specifying *msecInterval* as a number. Failure to do so will send the program into outer cyberspace when it executes because AtAddTimer is expecting an unsigned long value and not a 16-bit integer.

2.5.6 Example

The following C main routine opens an Altia interface connection, adds a timer callback to execute in 1 second, and then executes the toolkit main loop to wait for the timer event.

```
#include <stdio.h>
#include "altia.h"

/* Must have forward declaration for timerHandler so main can
 * reference it
 */
void timerHandler();

main(argc, argv)
int argc;
char *argv[];
{
    AtConnectId connectId;
    if ((connectId = AtOpenConnection(NULL, NULL, argc, argv)) < 0)
    {
        fprintf(stderr, "AtOpenConnection failed\n");
        exit(1);
    }

    /* For Windows 16-bit compilation, you must use the L notation on
    * msecInterval if it is a constant as shown below.
    */
    AtAddTimer(1000L, timerHandler, (AtPointer) connectId);
    AtMainLoop();
}

void timerHandler(data, interval, timerId)
AtPointer data;
unsigned long interval;
AtTimerId timerId;
{
    static AltiaEventType secCount = 0;
    /* Update an Altia animation function that is acting as a
    * counter display
    */
    AtSendEvent((AtConnectId) data, "timer", secCount++);
    /* Reinstate timer */
    AtAddTimer(interval, timerHandler, data);
}
```

2.5.7 See Also

AtMainLoop altiaSleep

2.6 AtCacheOutput, AtFlushOutput, altiaCacheOutput, altiaFlushOutput

2.6.1 Name

AtCacheOutput	toolkit function to turn event caching on or off.
altiaCacheOutput	base library version of AtCacheOutput.
AtFlushOutput	toolkit function to flush events that are cached.
altiaFlushOutput	base library version of AtFlushOutput.

2.6.2 C Synopsis

```
int AtCacheOutput(AtConnectId connectId, int onOff);

int altiaCacheOutput(int onOff);

int AtFlushOutput(AtConnectId connectId);

int altiaFlushOutput();
```

2.6.3 Description

Normally, each AtSendEvent/altiaSendEvent or AtSendText/altiaSendText call immediately transmits its data to the Altia interface and the interface updates the display graphics upon receiving the data. If, however, AtCacheOutput or altiaCacheOutput is called with *onOff* set to a non-zero value, events are cached (i.e., buffered) in memory maintained by the Altia library. The events are flushed to the Altia interface when the cache is filled, if AtFlushOutput or altiaFlushOutput is called, if AtCacheOutput or altiaCacheOutput is called with *onOff* set to zero, or when an Altia library routine is called that requires immediate communications with the Altia Interface (e.g., AtNextEvent or AtPollEvent require the interface to send back a response so the cache is flushed first). Finally, if events are cached as a result of calls by a callback function, the cache is flushed when the callback returns control to AtMainLoop.

Whenever the cache is flushed because it is full, the Altia interface accepts the events and records the graphics changes that should occur, but does not actually update the graphics display. The display is updated by calling AtFlushOutput or altiaFlushOutput, AtCacheOutput or altiaCacheOutput with *onOff* set to 0, or some Altia library routine that requires immediate communications with the Altia interface. If AtMainLoop is being used, the display is also updated on a return to AtMainLoop from a callback.

Turning on caching can significantly improve performance for graphics intensive animations. For example, plotting, strip charting, or extensive text output updating. It may also improve performance significantly when a network socket is being used for communication between the program and the Altia interface.

When caching is enabled from application code using AtCacheOutput/altiaCacheOutput, only events from the application are cached. Mouse, keyboard, and timer stimulus in Altia will still update the graphics display. When this happens, Altia also applies the cached events it has received from the application. Use the built-in animation function altiaCacheOutput to disable all display updates. For more information about this built-in function and many others, please see the ***Altia Design User's Guide***.

2.6.4 Return Values

If a communications failure occurs, these functions return -1. Otherwise, 0 is returned.

2.6.5 Windows Dependencies

To improve inter-process communications performance, events from AtSendEvent/altiaSendEvent and AtSendText/altiaSendText calls are automatically cached until a call that requires immediate communications with the Altia interface is executed (e.g., AtNextEvent or AtPollEvent) or the program returns to AtMainLoop from a callback if AtMainLoop is being used. If this behavior produces undesirable affects, a call to AtFlushOutput or altiaFlushOutput can be made to force cached events to be sent immediately. Or, turn off caching permanently with a call to AtCacheOutput or altiaCacheOutput.

2.6.6 See Also

AtSendEvent, altiaSendEvent, AtSendText, altiaSendText

2.7 AtCheckEvent, altiaCheckEvent

2.7.1 Name

AtCheckEvent	toolkit function to check for a pending event with a specific name.
altiaCheckEvent	base library version of AtCheckEvent.

2.7.2 C Synopsis

```
int AtCheckEvent(AtConnectId connectId, const char *eventName,
                 AltiaEventType *eventValueOut);

int altiaCheckEvent(const char *eventName, AltiaEventType *eventValueOut);
```

2.7.3 Description

Either function is called to get the value for the next available event that was delivered to the program by the Altia interface and whose name matches *eventName*. If there are no events available with the specified name, both functions return immediately.

If one or more events are queued with the specified name, the value for the oldest event is returned and that instance of the event is removed from the pending event queue. All other events, including newer instances of the named event, remain in the queue in their appropriate order. In order for these functions to work correctly, the event name must be selected with a call to AtSelectEvent, altiaSelectEvent, AtSelectAllEvents, or altiaSelectAllEvents prior to calling AtCheckEvent or altiaCheckEvent. Or, if AtAddCallback was called to register a callback for the event name, then it is automatically selected for receipt.

The *eventValueOut* parameter must point to an AltiaEventType variable. If an event is found, its value is returned in the AltiaEventType variable pointed to by *eventValueOut*. The type for *eventValueOut*, AltiaEventType, is an int if the integer libraries are used and a double if the floating point libraries are used.

Normally, events are received by calling AtNextEvent or altiaNextEvent. However, these functions get the next available event for any name and they block waiting for a new event if one is not immediately available. AtCheckEvent or altiaCheckEvent can be used in conjunction with AtNextEvent or altiaNextEvent, or by themselves to accommodate different program architectures. If a callback is registered for the event, a call to AtDispatchEvent with the event name and new value will dispatch the event to the callback.

AtCheckEvent is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaCheckEvent acts on the currently active Altia interface connection.

2.7.4 Return Values

If an event is found, 1 is returned and the variable pointed to by *eventValueOut* is set to the event's value. If no event is found, 0 is returned. If a communications failure occurs, -1 is returned.

2.7.5 See Also

AtSelectEvent, altiaSelectEvent AtNextEvent, altiaNextEvent, AtDispatchEvent

2.8 AtDispatchEvent

2.8.1 Name

AtDispatchEvent	Toolkit function to process callbacks for an event received with AtNextEvent.

2.8.2 C Synopsis

```
void AtDispatchEvent(AtConnectId connectId, const char *eventName,
                     AltiaEventType eventValue);
```

2.8.3 Description

This is a toolkit function to invoke callbacks for an Altia interface event that was received by an explicit AtNextEvent or altiaNextEvent call. In almost all cases, this routine would be called if altiaNextEvent or AtNextEvent fetches an event that the calling procedure doesn't know how to handle. AtDispatchEvent could then be called to dispatch the event and its value to any handlers that are registered for it (see AtAddCallback, AtRemoveCallback, AtCallbackProc for details on registering callbacks). AtDispatchEvent will not invoke timer callbacks registered with AtAddTimer or I/O callbacks registered with AtAddInput. AtMainLoop must be used to process time-outs or I/O callbacks.

The type for *eventValue*, AltiaEventType, is an int if the integer libraries are used and a double if the floating point libraries are used.

2.8.4 Return Values

None

2.8.5 Example

The following C routine shows how AtDispatchEvent can be used. The code is waiting for a specific event to occur - a press event. If events occur that are not of the desired type, they are given to AtDispatchEvent so they can be handled by callbacks if any are registered.

```
void waitforPress(connectId)
AtConnectId connectId;
{
    char *name;
    AltiaEventType value;
    int retVal;

    while ((retVal = AtNextEvent(connectId, &name, &value)) == 0 &&
           strcmp(name, "press") != 0)
        AtDispatchEvent(connectId, name, value);

    /* Must have the event we want or a communications error */
    if (retVal == 0)
        printf("Just received the event %s(%d)\n", name, (int) value);
}
```

2.8.6 See Also

AtMainLoop, AtNextEvent, altiaNextEvent, AtAddCallback, AtRemoveCallback, AtCallbackProc

2.9 AtEventSelected, altiaEventSelected

2.9.1 Name

AtEventSelected	Toolkit function to check if a specific event has been selected for receipt.
altiaEventSelected	Base library version of AtEventSelected.

2.9.2 C Synopsis

```
int AtEventSelected(AtConnectId connectId, const char *eventName);

int altiaEventSelected(const char *eventName);
```

2.9.3 Description

Either function is called to check if events of the given name have been selected for receipt by the program. Event names are selected for receipt using AtSelectEvent, altiaSelectEvent, AtSelectAllEvents, or altiaSelectAllEvents. An event name is also selected for receipt automatically when AtAddCallback is called to register a callback for the name. The *eventName* parameter points to the name of the event that should be checked.

These functions are useful for large, modular application programs. If one module wishes to determine if a particular event name has been selected for receipt by another module, it can call AtEventSelected or altiaEventSelected to do so.

AtEventSelected is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaEventSelected acts on the currently active Altia interface connection.

2.9.4 Return Values

If the event name is selected for receipt, 1 is returned. If the event name is not selected for receipt, 0 is returned. If a communications failure occurs, -1 is returned.

2.9.5 See Also

AtSelectEvent, altiaSelectEvent AtUnselectEvent, altiaUnselectEvent

2.10 AtGetPortName

2.10.1 Name

AtGetPortName	Toolkit function to get the name of the port associated with an open connection.

2.10.2 C Synopsis

```
const char *AtGetPortName(AtConnectId connectId);
```

2.10.3 Description

This toolkit routine returns a pointer to the port name for the given Altia interface connection. The port name is the basis for the socket, DDE, or pipe name used to establish the connection. It could be used by another application as the portName parameter to an AtOpenConnection, AtStartInterface, or altiaConnect call if the application wished to connect to the same Altia interface session.

The string returned by AtGetPortName is the internal copy kept by the toolkit library. It should *NOT* be modified in any way. If modification is necessary, a copy of it should be made by the calling routine.

2.10.4 Return Values

A pointer to a character string is returned upon successful completion. The string pointed to is the internal copy kept by the toolkit library. It should *NOT* be modified in any way. If *connectId* refers to an invalid connection, NULL is returned.

2.10.5 See Also

AtOpenConnection, AtCloseConnection, AtStartInterface, altiaStartInterface, AtStopInterface, altiaFetchArgcArgv

2.11 AtGetText, altiaGetText

2.11.1 Name

AtGetText	Toolkit function to get a text string from an Altia text i/o object.
altiaGetText	Base library version of AtGetText.

2.11.2 C Synopsis

```
int AtGetText(AtConnectId connectId, const char *textName, char *buffer,
              int bufferSize)

int altiaGetText(const char *textName, char *buffer, int bufferSize);
```

2.11.3 Description

Both functions get the text string currently being displayed by a text i/o object.

First, some background on Altia text i/o objects is in order. Altia text i/o objects allow Altia interface designs to accept keyboard input from users. The input is displayed in a text field and dynamically updated as the user types in characters from the computer's keyboard. Text i/o objects have a built-in animation function, referred to as the "text" animation, which can be queried at any time by an application program. A query causes the text i/o object to send its stored text as a string of "text" events. The events can be received by the program and stored in a character buffer. Several styles of text i/o objects are available from the Inputs, Textio, and GUI Models Libraries. These are design files, inputs.dsn, textio.dsn, and gui.dsn or guiwin.dsn to be precise, found in the models sub-directory of the Altia software installation. They are organized for display within an Altia Models View (press the **Libraries** button in the Altia Design Editor to initiate the opening of a Models View). To learn specific details about text i/o objects, select the **Textio Models** option from the **Help** menu found in any Altia Models View.

AtGetText and altiaGetText query the "text" animation that is specified by the string pointed to by *textName*. The name could simply be "text" or it could be something else if the original object's "text" animation was renamed using the Altia Animation Rename Dialog. The text that is retrieved is stored in the character buffer pointed to by *buffer* as a null terminated string (i.e., ends in '\0'). Note that the caller must supply a buffer - it is *not* supplied by the routines. The character size of the buffer is specified by *bufferSize*. If the string returned by the object is larger than *bufferSize*, then it is truncated and null terminated to fit in the character buffer.

AtGetText is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaGetText acts on the currently active Altia interface connection.

Designs created prior to software release 2.0 could only use the obsoleted (but still supported) text input object instead of the newer and more versatile text i/o object. The text input object has a "getchar" animation instead of a "text" animation. It is queried using AtGetText in the same way as the "text" animation.

2.11.4 Return Values

If a communications failure occurs or the "text" animation specified by *textName* does not respond to a query, these functions return -1. In either case, the first element of the array pointed to by *buffer* is set to the null character ('\0'). A 0 is returned if the query was successful and the array pointed to by *buffer* will contain a null terminated string. If the input i/o object is not displaying any text, the array will only contain a null character.

2.11.5 Example

The following C code shows how AtGetText can be used with one of the text input areas from the Inputs, Textio, or GUI Models Libraries. For software release 2.0 or later, text input areas from these libraries generate a "done" event when a user finishes text input. Text input is finished by pressing the Return key or by clicking the left mousebutton outside of the input area. This example registers a callback for the "done" event. When the callback is invoked, it queries the "text" animation for the text i/o object to get the user's text input.

```
#include "altia.h"

/* Must have forward declaration for doneHandler so
 * main can reference it
 */
void doneHandler();
main(argc, argv)
int argc;
char *argv[];
{
    AtConnectId connectId;
    if ((connectId = AtOpenConnection(NULL, NULL, argc, argv)) < 0)
    {
        fprintf(stderr, "AtOpenConnection failed\n");
        exit(1);
    }
    AtAddCallback(connectId, "done", doneHandler, NULL);
    AtMainLoop();
}

void doneHandler(connectId, eventName, eventValue, data)
AtConnectId connectId;
char *eventName;
AltiaEventType eventValue;
AtPointer data;
{
    char buffer[50];
    /* This code assumes there is a second text i/o object
    * in the interface with a "text" animation named "out_text"
    * it sends the input text to it or an error message.
    */
    if (AtGetText(connectId, "text", buffer, 50) == 0)
        AtSendText(connectId, "out_text", buffer);
    else
        AtSendText(connectId, "out_text", "Error - AtGetText failed");
}
```

2.11.6 See Also

AtAddTextCallback, AtRemoveTextCallback, AtTextProc

2.12 AtInputNumber, AtOutputNumber, altiaInputNumber, altiaOutputNumber

2.12.1 Name

AtInputNumber	Toolkit function to get the input file descriptor for an Altia interface connection.
altiaInputNumber	Base library version of AtInputNumber.
AtOutputNumber	Toolkit function to get the output file descriptor for an Altia interface connection.
altiaOutputNumber	Base library version of AtOutputNumber.

2.12.2 C Synopsis

```
int AtInputNumber(AtConnectId connectId);

int altiaInputNumber();

int AtOutputNumber(AtConnectId connectId);

int altiaOutputNumber();
```

2.12.3 Description

AtInputNumber and altiaInputNumber return the input descriptor for an Altia interface connection. On UNIX based systems, this is the file descriptor for the input socket or pipe used for communications between the application process and the Altia interface process. This file descriptor is most commonly used with the select(2) UNIX system call to simultaneously wait for input from multiple sources. To use the Altia connection file descriptor with select(2), the application should first call AtPending/ altiaPending to get the number of available events, process exactly that number of events with calls to AtNextEvent/altiaNextEvent, then use select(2) to wait for more Altia interface events and input from other sources. It is very important that all known pending Altia events get processed before select(2) is called. Otherwise, select(2) will block even though Altia events are available because the events are already queued and will not force the select(2) out of its wait.

AtInputNumber is the toolkit version of the altiaInputNumber function. It takes a connection id as returned by AtOpenConnection or AtStartInterface.

AtOutputNumber and altiaOutputNumber return the output descriptor for an Altia interface connection. On UNIX based systems, this is the file descriptor for the output socket or pipe used for communications between the application process and the Altia interface process. Frankly, its hard to think of any reason an application program would need to access the output descriptor. Even so, we have provided a way to get it for consistency reasons.

AtOutputNumber is the toolkit version of the altiaOutputNumber function. It takes a connection id as returned by AtOpenConnection or AtStartInterface.

Use of any of these functions is not usually necessary for the typical applications program, especially if one is using AtMainLoop to control program flow. Under this situation, it is possible to add callbacks for other input sources using AtAddInput and AtMainLoop will handle the calling of select(2) for all input sources.

2.12.4 Return Values

A file descriptor is returned upon successful completion. If no connection is established, -1 is returned.

2.12.5 Windows Dependencies

These functions always return -1 for the libdde.lib version of the Altia library. A DDE does not have a file descriptor number.

2.12.6 Example

The following C function demonstrates the use of AtInputNumber for processing Altia interface events and X11 events simultaneously. The function assumes that a valid Altia connection id and X11 display pointer are being provided as parameters.

```
int processEvents(display, connectId)
AtConnectId connectId;
Display     *display;
{
    int fd1, fd2, xCount, altiaCount, maxFD;
    fd_set readfds, writefds, exceptfds;
    int retval = 0;
    char *name;
    AltiaEventType value;
    XEvent event;

    /* Get X input file descriptor, then Altia input file descriptor */

    fd1 = ConnectionNumber(display);
    fd2 = AtInputNumber(connectId);

    /* Clear various masks */
    FD_ZERO(&readfds);
    FD_ZERO(&writefds);
    FD_ZERO(&exceptfds);

    /* Now set masks for fd1 and fd2 */
    maxFD = fd1;
    FD_SET(fd1, &readfds);
    if (maxFD < fd2)
        maxFD = fd2;
    FD_SET(fd2, &readfds);

    /* If events are already pending, no need to block */

    xCount = XPending(display);
    altiaCount = AtPending(connectId);

    if (xCount == 0 && altiaCount == 0)
    {
        /* If no events pending, then use select(2) to block
        * waiting for an event from either source.
        */
        retval = select(maxFD + 1,
            (fd_set *) &readfds, (fd_set *) &writefds, (fd_set *)
            &exceptfds, NULL);
```

```
            /* If X caused us to return, set xCount to 1.
            * Could also call XPending() to get a count.
            */
            if (FD_ISSET(fd1, &readfds))
                xCount = 1;

            /* If Altia caused us to return, set altiaCount to 1.
            * Could call altiaPending() to get a count.
            */
            if (FD_ISSET(fd2, &readfds))
                altiaCount = 1;
        }
        /* If we encountered an error, return -1 now.
        * Customize this error handling as is appropriate.
        */
        if (xCount < 0 || altiaCount < 0 || retval < 0)
            return -1;

        /* Process 1 or more X events. Assuming X events take priority
        * over Altia events and we want to process all that are
        * available.These assumptions may not be appropriate for all
        * apps.
        */
        while (xCount-- > 0)
        {
            XNextEvent(display, &event);
            /* Do original X event processing here */
        }

        /* Process 1 or more Altia events. Assuming that Altia callbacks
        * are being used, this loop is quite simple. If Altia callbacks
        * are not being used, then there would be no call to
        * AtDispatchEvent() and instead the returns from AtNextEvent()
        * would be processed directly.Please note that Altia timer
        * callbacks (i.e. callbacks set up using the AtAddTimer()
        * function) do not work when we get and dispatch events
        * manually.
        */
        while (altiaCount-- > 0)
        {
            AtNextEvent(connectId, &name, &value);
            AtDispatchEvent(connectId, name, value);
        }
        return 0;
    }
```

2.12.7 See Also

AtMainLoop, AtAddInput, AtRemoveInput, AtInputProc

2.13 AtLocalPending, altiaLocalPending

2.13.1 Name

AtLocalPending	Toolkit function to get the number of Altia interface events waiting to be received locally.
altiaLocalPending	Base library version of AtPending.

2.13.2 C Synopsis

```
int AtLocalPending(AtConnectId connectId);

int altiaLocalPending();
```

2.13.3 Description

Both functions check for local events that are waiting to be received. The number of events available is returned where a return of zero (0) indicates no events are available.

AtLocalPending is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaLocalPending acts on the currently active Altia interface connection.

In contrast, AltiaPending/AtPending will check the Altia interface process for additional events if none are available locally.

Normally, a call to AtLocalPending or altiaLocalPending is made prior to a AtNextEvent/ altiaNextEvent call to determine if there are any local events to get.

2.13.4 Return Values

Zero (0) is returned if no local events are pending. A positive non-zero value is returned if events are pending and the value reflects the number of pending events. If a communications failure occurs, -1 is returned.

2.13.5 See Also

AtNextEvent, altiaNextEvent AtLocalPollEvent, altiaLocalPollEvent

2.14 AtLocalPollEvent, altiaLocalPollEvent

2.14.1 Name

AtLocalPollEvent	Toolkit function to get the current state of an Altia interface animation event.
altiaLocalPollEvent	Base library version of AtLocalPollEvent.

2.14.2 C Synopsis

```
int AtLocalPollEvent(AtConnectId connectId, const char *eventName,
                     AltiaEventType *eventValueOut);

int altiaLocalPollEvent(const char *eventName, AltiaEventType *eventValueOut);
```

2.14.3 Description

Both functions get the current state of a local animation event. The event's name is given by eventName and its current state is returned in the variable pointed to by eventValueOut. The type for eventValueOut, AltiaEventType, is an int if the integer libraries are used and a double if the floating point libraries are used.

If any events with the same name were previously routed to this client application and they are awaiting receipt via AtNextEvent, altiaNextEvent or AtMainLoop, the events will be deleted from the local receiving queue. A poll request gets the current local value for the event - this makes all previous values obsolete.

AtLocalPollEvent is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaLocalPollEvent acts on the currently active Altia interface connection.

2.14.4 Return Values

If there is no local animation event with the given name or a communications failure occurs, these functions return -1. Otherwise, 0 is returned.

2.14.5 See Also

AtPollEvent, altiaPollEvent, AtNextEvent, altiaNextEvent, AtSelectEvent, altiaSelectEvent

2.15 AtMainLoop

2.15.1 Name

AtMainLoop	Main processing loop to manage toolkit event, timer, and I/O callbacks.

2.15.2 C Synopsis

```
void AtMainLoop();
```

2.15.3 Description

AtMainLoop loops indefinitely, processing pending Altia interface events, timer events, and I/O events. It uses the select(2) system call, or its equivalent, to suspend program execution until some event is ready for processing. This greatly reduces the CPU usage by the program. Event processing entails identifying pending events and invoking the appropriate callback procedures that have been registered with calls to AtAddCallback, AtAddTextCallback, AtAddCloneCallback, AtAddTimer, and AtAddInput. AtMainLoop will typically never return control to its calling procedure unless a fatal select(2) system call error occurs or all Altia interface connections have closed and there are no timer or I/O event callbacks registered.

Typically, AtMainLoop is called from main after all callbacks have been registered and variables, file descriptors, etc. have been initialized.

Much of the functionality of AtMainLoop can be duplicated by a custom main loop using the functions AtNextEvent and AtDispatchEvent. Timer and input callbacks, however, require the use of AtMainLoop. See AtDispatchEvent for more details.

2.15.4 Return Values

None

2.15.5 Examples

Manual pages for AtAddCallback, AtAddTextCallback, AtAddTimer, and AtAddInput provide examples for AtMainLoop usage.

2.15.6 See Also

AtAddCallback, AtRemoveCallback, AtCallbackProc, AtAddTextCallback, AtRemoveTextCallback, AtTextProc, AtAddTimer, AtRemoveTimer, AtTimerProc, AtAddInput, AtRemoveInput, AtInputProc, AtAddCloneCallback, AtRemoveCloneCallback, AtCloneProc, AtDispatchEvent

2.16 AtMoveObject, altiaMoveObject

2.16.1 Name

AtMoveObject	Toolkit function to reposition an object within an Altia interface.
altiaMoveObject	Base library version of AtMoveObject.

2.16.2 C Synopsis

```
int AtMoveObject(AtConnectId connectId, int objectNumber, int xOffset,
                 int yOffset);

int altiaMoveObject(int objectNumber, int xOffset, int yOffset);
```

2.16.3 Description

Both functions move the Altia interface object identified by *objectNumber* from its current position within the interface to a new position. The new position is *xOffset* pixels from the original position in the x direction and *yOffset* pixels from the original position in the y direction (i.e., the move is relative to the object's current position). *xOffset* or *yOffset* is positive to achieve a move to the right or up, respectively. *xOffset* or *yOffset* is negative to achieve a move to the left or down, respectively. All animation and stimulus for the object will be performed relative to the object's new position. Note that move animation can be used to move an object rather than using these Altia library calls. The calls, however, allow for more freedom of movement.

An object's object number can be determined from the Altia Design Editor interface by selecting the object and then choosing the **Rename Animation** item from the Animation Editor's or Stimulus Editor's **Edit** pulldown menu. This will display the Function Rename Dialog. The **Prepend** field of the Rename Dialog displays the object's object number.

AtMoveObject is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaMoveObject acts on the currently active Altia interface connection.

2.16.4 Return Values

If a communications failure occurs, these functions return -1. Otherwise, 0 is returned. If *objectNumber* is invalid, 0 may still be returned because the functions are implemented using asynchronous communication. As a result, the Altia interface doesn't have a chance to return an error value to the application if *objectNumber* is invalid.

2.17 AtNextEvent, altiaNextEvent

2.17.1 Name

AtNextEvent	Toolkit function to get the next available event from an Altia interface.
altiaNextEvent	Base library version of AtNextEvent.

2.17.2 C Synopsis

```
int AtNextEvent(AtConnectId connectId, char **eventNameOut,
                AltiaEventType *eventValueOut);

int altiaNextEvent(char **eventNameOut, AltiaEventType *eventValueOut);
```

2.17.3 Description

Both functions get the next available event that was routed to the client application by an Altia interface. The functions "block" until an event is available. The only events that are routed are those that have been previously selected by calling AtSelectEvent/altiaSelectEvent or AtSelectAllEvents/altiaSelectAllEvents. The parameter *eventNameOut* must be a pointer to a character pointer and *eventValueOut* must be a pointer to an AltiaEventType type variable. AltiaEventType is an int if the integer libraries are used and a double if the floating point libraries are used.

Upon successful completion, the character pointer pointed to by *eventNameOut* will point to a string containing the name of the newly acquired event. The integer pointed to by *eventValueOut* will contain the value for the event. The string that is returned resides in memory that is private to the library. BE CAREFUL! This memory may be overwritten by the next call to any function in the library.

AtNextEvent is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaNextEvent acts on the currently active Altia interface connection.

Normally, incoming events received by AtNextEvent or altiaNextEvent are originally generated by the execution of stimulus definitions within the Altia interface. That is, a user manipulates the mouse or keyboard in such a way that a previously defined stimulus definition is executed.

2.17.4 Return Values

Zero (0) is returned upon successful completion. If a communications failure occurs or the client has no events selected for routing, -1 is returned and the contents of *eventNameOut* and *eventValueOut* are not deterministic.

2.17.5 Example

The following C main routine opens an Altia interface connection, selects to receive **press** and **slide** events, and then calls AtNextEvent to wait for instances of the selected events.

```
#include <string.h>
#include "altia.h"
main(argc, argv)
```

```
int argc;
char *argv[];
{
    AtConnectId connectId;
    char *newName;
    AltiaEventType newValue;
    if ((connectId = AtOpenConnection(NULL, NULL, argc, argv)) < 0)
    {
        fprintf(stderr, "AtOpenConnection failed\n");
        exit(1);
    }
    AtSelectEvent(connectId, "press");
    AtSelectEvent(connectId, "slide");
    while (AtNextEvent(connectId,&newName, &newValue) == 0)
    {
        if (strcmp(newName, "press") == 0)
            printf("Received event press(%d)\n", (int) newValue);
        else if (strcmp(newName, "slide") == 0)
            printf("Received event slide(%d)\n", (int) newValue);
    }
}
```

2.17.6 See Also

AtPollEvent, altiaPollEvent, AtPending, altiaPending, AtSelectEvent, altiaSelectEvent, AtAddCallback, AtRemoveCallback, AtCallbackProc, AtDispatchEvent

2.18 AtOpenConnection, AtCloseConnection

2.18.1 Name

AtOpenConnection	Toolkit function to open a connection to an Altia interface.
AtCloseConnection	Toolkit function to close an Altia interface connection.

2.18.2 C Synopsis

```
AtConnectId AtOpenConnection(const char *portName, const char *optionName,
                             int argc, const char *argv[]);

void AtCloseConnection(AtConnectId connectId);
```

2.18.3 Description

AtOpenConnection is a toolkit function to open a connection to an Altia interface. This function returns a connection id that serves as a parameter to most other toolkit routines. The *portName* parameter is analogous to the *portName* parameter for altiaConnect. The *optionName* parameter is explained in further detail in the next paragraph. The *argv* parameter is an array of string pointers. It allows for the passing of additional arguments as strings. The number of elements in *argv* is specified by *argc*. Typically, *argc* and *argv* are the argument count and list passed to the program's main routine - these are arguments that originated as command line options when the program was initially executed.

If *portName* is non-NULL, it must point to a string containing a name to use for the Altia interface connection port name. If *portName* and *optionName* are NULL, and an argument pair of the form **-port** *PORTNAME* is found in the *argv* list, then *PORTNAME* is used to construct the port name. If *optionName* is not NULL, the string it points to is used as the option differentiator instead of **-port** when searching the *argv* list. For example, if *portName* is NULL and *optionName* points to the string "**-screen**", then the *argv* argument pair **-screen /tmp/screen** would set the connection port name to /tmp/screen. If *portName* is NULL and no port arguments are found in the *argv* list, AtOpenConnection opens the default connection if it is not already opened. Only one Altia interface can operate on the default connection at any given time on a single workstation. To open a connection to an alternative Altia interface, one must specify a port name.

For UNIX systems, a domain or network socket serves as the connection facility between an Altia interface and its client application programs when application programs are linked with the liblan.a program library. A domain socket is specified by using a filename syntax for the port name (e.g., /tmp/screen). A network socket is specified by using a *HOSTNAME:SOCKET* syntax where *SOCKET* may be a number or it may be an alias name for a socket number as specified in the /etc/services or equivalent file. If *HOSTNAME* is omitted, then the name of the current host is assumed.

On Microsoft Windows systems, liblan.lib only supports network sockets and they are specified using the syntax described above. Windows programs may also use a DDE mechanism for client/interface communication by linking with libdde.lib. The DDE method is the only choice for systems that have no TCP/IP sockets software or hardware installed. The *portName* argument or a port name parsed from the command line is simply a logical name of your choosing for the DDE Service Name. For example, "AltDDE" is used as the default DDE Service Name if none is specified.

In cases where AtOpenConnection is provided a port name (either through *portName* or the *argv* list) and the program is linked with liblan.a (liblan.lib on Windows), the Altia interface that the client is connecting to must be started with the command line argument **-port** *PORTNAME* (**-lan -port** *PORTNAME* on Windows 3.1 and 9x) where *PORTNAME* matches the connection port name parsed by AtOpenConnection. If AtOpenConnection isn't given a specific port name, it opens a default domain socket port (/usr/tmp/vSe.hostname) for UNIX systems or a default network socket port (5100) for Windows systems. In these cases, the **-port** *PORTNAME* argument is unnecessary for the Altia interface start-up because it will open the same socket by default, but the **-lan** option is still necessary on Windows 3.1 and 9x systems for proper socket initialization to occur.

In cases where AtOpenConnection is provided a port name (either through *portName* or the *argv* list) and the program is linked with libdde.lib on a Windows system, the Altia interface that the client is connecting to must be started with the command line argument **-dde** *PORTNAME* where *PORTNAME* matches the connection port name parsed by AtOpenConnection. If AtOpenConnection isn't given a specific port name, it uses a default DDE Service Name (AltDDE to be precise). In this case, the **-dde** *PORTNAME* argument is unnecessary for the Altia interface start-up because it always opens the same DDE Service by default.

AtOpenConnection also parses the *argv* list for the argument pair **-retry** *COUNT* to set the count for retrying a failed connection. If no retry specification is found, the retry count is set to 1. A call to AtRetryCount can also be used to set the retry count to something other than 1.

Finally, AtOpenConnection also parses the *argv* list for the argument pair **-debug** *LEVEL* where *LEVEL* is one of the values 0, 1, or 2. This turns on different levels of debug. Level 0 is debug off. In level 1 debug, messages are sent to **stderr** each time a connection is opened or closed, a callback is added or removed, or an event is received. Level 2 debug includes level 1 messages and adds messages for each time the program sends an event to an Altia interface using AtSendEvent or AtSendText.

NOTE: AtOpenConnection can only succeed if there is an Altia interface already running that is *serving* the selected domain or network socket or DDE. To start an interface directly from a program, use AtStartInterface.

AtCloseConnection closes a connection that was opened with AtOpenConnection or AtStartInterface. It is not usually necessary to explicitly close a connection. The connection will close automatically when the program exits. The Altia interface is not affected by a connection close by the client - it will continue to execute. AtStopInterface, on the other hand, will close the connection and attempt to stop the Altia interface. It does so by sending an altiaQuit event to the interface. This will normally stop the interface unless the quit event name has been changed by a Altia*quitFunction: resource specification. The specification can come from a Runtime Configuration File (*.rtm* file), *X* resource database, *X* defaults file, or Altia applications defaults file.

2.18.4 Return Values

AtOpenConnection returns a non-negative connection id if it was successful. If a connection could not be established, -1 is returned. AtCloseConnection has no return value.

2.18.5 Examples

The following C main routine opens the default Altia interface connection or it accepts a **-port** *PORTNAME* command line argument. It sends an event to the interface and then closes the connection. This code will only work if an Altia interface is already running and is *serving* clients on the same port.

```
#include "altia.h"
main(argc, argv)
int argc;
char *argv[];
{
    AtConnectId connectId;
    if ((connectId = AtOpenConnection(NULL, NULL, argc, argv)) < 0)
    {
        fprintf(stderr, "AtOpenConnection failed\n");
        exit(1);
    }
    AtSendEvent(connectId, "start", (AltiaEventType) 1);
    AtCloseConnection(connectId);
}
```

The next C main routine opens two Altia interface connections. The caller needs to pass port names on the command line using **-screen** *SCREENPORT* and **-panel** *PANELPORT*. It sends an event to each interface and then exits. This code will only work correctly if two Altia interfaces are already running and one was started with **-port** *SCREENPORT (***-lan -port** *SCREENPORT* on Windows 3.1 and 9x) arguments and the other with **-port** *PANELPORT* (**-lan -port** *PANELPORT* on Windows 3.1 and 9x) arguments. On Windows systems, replace **-port** or **-lan -port** with **-dde** if the program is linked with libdde.lib.

```
#include "altia.h"
main(argc, argv)
int argc;  char *argv[];
{
    AtConnectId screenId, panelId;
    if ((screenId = AtOpenConnection(NULL, "-screen", argc, argv)) < 0)
    {
        fprintf(stderr, "AtOpenConnection for screen failed\n");
        exit(1);
    }
    if ((panelId = AtOpenConnection(NULL, "-panel", argc, argv)) < 0)
    {
        fprintf(stderr, "AtOpenConnection for panel failed\n");
        exit(1);
    }
    AtSendEvent(panelId, "start", (AltiaEventType) 1);
    AtSendEvent(screenId, "start", (AltiaEventType) 1);
    AtCloseConnection(panelId);
    AtCloseConnection(screenId);
}
```

2.18.6 See Also

AtStartInterface, altiaStartInterface, AtStopInterface, altiaFetchArgcArgv, AtRetryCount, altiaRetryCount, altiaConnect, altiaDisconnect, altiaStopInterface, altiaSuppressErrors

2.19 AtPending, altiaPending

2.19.1 Name

AtPending	Toolkit function to get the number of Altia interface events waiting to be received.
altiaPending	Base library version of AtPending.

2.19.2 C Synopsis

```
int AtPending(AtConnectId connectId);

int altiaPending();
```

2.19.3 Description

Both functions check for Altia interface events that are waiting to be received. The number of events available is returned where a return of zero (0) indicates no events are available. Events are selected for receipt using AtSelectEvent/altiaSelectEvent or AtSelectAllEvents/altiaSelectAllEvents.

AtPending is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaPending acts on the currently active Altia interface connection.

Normally, a call to AtPending or altiaPending is made prior to an AtNextEvent/altiaNextEvent call to determine if there are any events to get. This is especially true if the program wants to avoid blocking in the AtNextEvent/altiaNextEvent call. Instead, it may wish to do other processing and come back later for another check.

2.19.4 Return Values

Zero (0) is returned if no events are pending. A positive non-zero value is returned if events are pending and the value reflects the number of pending events. If a communications failure occurs, -1 is returned.

2.19.5 Example

The following C function checks for an Altia interface event, gets a new event if one is available, and dispatches the event using AtDispatchEvent assuming a callback is registered for the event. It returns the count of pending events or -1 if there was a communications failure.

```
int checkForEvent(id)
AtConnectId id;
{
    char *name;  AltiaEventType value;  int retVal;
    if ((retVal = AtPending(id)) > 0)
    {
        AtNextEvent(id, &name, &value);
        AtDispatchEvent(id, name, value);
    }
    return retVal;
}
```

2.19.6 See Also

AtNextEvent, altiaNextEvent, AtPollEvent, altiaPollEvent, AtSelectEvent, altiaSelectEvent

2.20 AtPollEvent, altiaPollEvent

2.20.1 Name

AtPollEvent	Toolkit function to get the current state of an Altia interface animation event.
altiaPollEvent	Base library version of AtPollEvent.

2.20.2 C Synopsis

```
int AtPollEvent(AtConnectId connectId, const char *eventName,
                AltiaEventType *eventValueOut);

int altiaPollEvent(const char *eventName, AltiaEventType *eventValueOut);
```

2.20.3 Description

Both functions get the current state of an Altia interface animation event. The event's name is given by *eventName* and its current state is returned in the variable pointed to by *eventValueOut*. The type for *eventValueOut*, AltiaEventType, is an int if the integer libraries are used and a double if the floating point libraries are used.

If any events with the same name were previously routed to this client application and they are awaiting receipt via AtNextEvent, altiaNextEvent or AtMainLoop, the events will be deleted from the receiving queue. A poll request gets the current value for the event - this makes all previous values obsolete.

AtPollEvent is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaPollEvent acts on the currently active Altia interface connection.

2.20.4 Return Values

If there is no Altia interface animation event with the given name or a communications failure occurs, these functions return -1. Otherwise, 0 is returned.

2.20.5 Example

The following C main routine opens a connection and then uses AtPollEvent to wait for a non-zero value for event "power".

```
#include "altia.h"
main(argc, argv)
int argc;
char *argv[];
{
    AtConnectId connectId;
    AltiaEventType newValue;
    if ((connectId = AtOpenConnection(NULL, NULL, argc, argv)) < 0)
    {
        fprintf(stderr, "AtOpenConnection failed\n");
        exit(1);
    }

    while (AtPollEvent(connectId, "power", &newValue) != 0 || newValue == 0)
```

```
        {
            /* Waiting for power button press so we
             * can do some initialization
             */
            altiaSleep(1000L);
        }

        /* More code would go here */
    }
```

2.20.6 See Also

AtSendEvent, altiaSendEvent, AtNextEvent, altiaNextEvent, AtSelectEvent, altiaSelectEvent

2.21 AtRemoveAllCallbacks

2.21.1 Name

AtRemoveAllCallbacks	Removes all types of callbacks registered for a specific Altia interface event.

2.21.2 C Synopsis

```
void AtRemoveAllCallbacks(AtConnectId connectId, const char *eventName);
```

2.21.3 Description

This toolkit routine removes all procedures from the Altia interface event callback list that are registered for the event name specified by *eventName* and are associated with the Altia interface connection *connectId*. This routine will remove all normal callbacks, text callbacks, and clone callbacks that have been registered for the event. To reduce event traffic and improve performance, AtRemoveAllCallbacks calls AtUnselectEvent to request that the Altia interface no longer send *eventName* events to the application.

2.21.4 Return Values

None

2.21.5 See Also

AtAddCallback, AtRemoveCallback, AtCallbackProc, AtAddTextCallback, AtRemoveTextCallback, AtTextProc, AtAddCloneCallback, AtRemoveCloneCallback, AtCloneProc

2.22 AtRetryCount, altiaRetryCount

2.22.1 Name

AtRetryCount	Toolkit function to set the count for retrying an Altia interface connection after a failure
altiaRetryCount	Base library version of AtRetryCount.

2.22.2 C Synopsis

```
void AtRetryCount(AtConnectId connectId, int count);

void altiaRetryCount(int count);
```

2.22.3 Description

Under default conditions, an Altia interface connection opened with altiaConnect or AtStartInterface allows communications failures to occur without bounds (i.e., an infinite retry count). An Altia interface connection opened with the toolkit function AtOpenConnection allows no communications failures (i.e., a retry count of 1). If a toolkit or base library call fails, the library calls *exit* to terminate the program.

In any case, the retry count can be changed to suit the application. If the connection was opened with altiaConnect, then call altiaRetryCount. If the connection was opened with AtOpenConnection or AtStartInterface, then call AtRetryCount. Given a *count* parameter of value *N*, the library will call *exit* when the *N*-th call to a library function fails. To select an infinite retry, pass a *count* of 0.

2.22.4 Return Values

None

2.22.5 See Also

AtOpenConnection, AtCloseConnection, AtStartInterface, altiaStartInterface, AtStopInterface, altiaFetchArgcArgv, altiaConnect, altiaDisconnect, altiaStopInterface

2.23 AtSelectAllEvents, altiaSelectAllEvents

2.23.1 Name

AtSelectAllEvents	Toolkit function to select to receive all events from an Altia interface.
altiaSelectAllEvents	Base library version of AtSelectAllEvents.

2.23.2 C Synopsis

```
int AtSelectAllEvents(AtConnectId connectId);

int altiaSelectAllEvents();
```

2.23.3 Description

Either function is called to select to have *all* types of Altia interface events routed to the application. Events can actually be received by calling AtNextEvent / altiaNextEvent.

AtSelectAllEvents is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaSelectAllEvents acts on the currently active Altia interface connection.

It is not recommended that AtSelectAllEvents or altiaSelectAllEvents be used to shortcut the specific selection of events with AtSelectEvent / altiaSelectEvent. Selecting to receive all events may degrade application program and Altia interface performance because many events generated by the Altia interface may only be of interest to the Altia interface itself. For example, stimulus enable conditions may not be of any interest to the application program; however, they may be generated quite often depending upon the architecture of the interface.

2.23.4 Return Values

Zero (0) is returned upon successful completion. If a communications failure occurs, -1 is returned.

2.23.5 See Also

AtSelectEvent, altiaSelectEvent, AtUnselectEvent, altiaUnselectEvent, AtUnselectAllEvents, altiaUnselectAllEvents, AtNextEvent, altiaNextEvent, AtPending, altiaPending

2.24 AtSelectEvent, altiaSelectEvent

2.24.1 Name

AtSelectEvent	Toolkit function to select to receive a particular type of Altia interface event.
altiaSelectEvent	Base library version of AtSelectEvent.

2.24.2 C Synopsis

```
int AtSelectEvent(AtConnectId connectId, const char *eventName);

int altiaSelectEvent(const char *eventName);
```

2.24.3 Description

Either function is called to select to have a specific type of event routed to the application by an Altia interface. Events that are selected in this way can be received by calling AtNextEvent / altiaNextEvent. The *eventName* parameter specifies the name of the event that the application now wishes to receive. This would be in addition to any other events that have been selected for receipt with prior calls to AtSelectEvent or altiaSelectEvent. An application may also select to receive events of all types by calling AtSelectAllEvents / altiaSelectAllEvents.

AtSelectEvent is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaSelectEvent acts on the currently active Altia interface connection.

NOTE: It is unnecessary to call AtSelectEvent for events that have or will have callbacks associated with them. The function for adding a callback for an event, AtAddCallback, automatically selects to receive events of the specified type.

2.24.4 Return Values

Zero (0) is returned upon successful completion. If a communications failure occurs, -1 is returned.

2.24.5 See Also

AtUnselectAllEvents, altiaUnselectAllEvents, AtSelectAllEvents, altiaSelectAllEvents, AtNextEvent, altiaNextEvent, AtPending, altiaPending

2.25 AtSendEvent, altiaSendEvent

2.25.1 Name

AtSendEvent	Toolkit function to send an animation event to an Altia interface.
altiaSendEvent	Base library version of AtSendEvent.

2.25.2 C Synopsis

```
int AtSendEvent(AtConnectId connectId, const char *eventName,
                AltiaEventType eventValue);

int altiaSendEvent(const char *eventName, AltiaEventType eventValue);
```

2.25.3 Description

Both functions initiate the transmission of an animation event to an Altia interface. The event's name is given by *eventName* and its value is given by *eventValue*. The type for *eventValue*, AltiaEventType, is an int if the integer libraries are used and a double if the floating point libraries are used.

AtSendEvent is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaSendEvent acts on the currently active Altia interface connection.

A call to AtSendEvent or altiaSendEvent forces a state change of an Altia interface animation function. This usually results in a visual change to one or more objects within the Altia interface.

2.25.4 Return Values

If a communications failure occurs, these functions return -1. Otherwise, 0 is returned.

2.25.5 Windows Dependencies

To improve inter-process communications performance, events from AtSendEvent and altiaSendEvent calls are buffered on the application's side until a synchronous call such as AtNextEvent/altiaNextEvent or AtPollEvent/altiaPollEvent is performed or the program returns to AtMainLoop from a callback if AtMainLoop is being used. If this behavior produces undesirable effects, a call to AtFlushOutput/altiaFlushOutput can be made to force buffered events to be sent immediately. Or, turn off buffering permanently with a call to AtCacheOutput/altiaCacheOutput.

2.25.6 See Also

AtSendText, altiaSendText, AtCacheOutput, AtFlushOutput, altiaCacheOutput, altiaFlushOutput

2.26 AtSendText, altiaSendText

2.26.1 Name

AtSendText	Toolkit function to send a string of animation events to an Altia interface.
altiaSendText	Base library version of AtSendText.

2.26.2 C Synopsis

```
int AtSendText(AtConnectId connectId, const char *eventName, const char *text);

int altiaSendText(const char *eventName, const char *text);
```

2.26.3 Description

Both functions initiate the transmission of a string of animation events to an Altia interface. The values for the events are the values of the characters provided by the string pointed to by *text* and must be terminated by the null character ('\0'). The values, including the terminating null character, are sent to the animation function given by *eventName*.

AtSendText and altiaSendText are normally used for transmitting character strings to Altia interface text i/o objects. Altia text i/o objects have built-in animation, referred to as the "text" and "character" animations, that accept events and treat them as text characters. The characters are displayed on the screen in the font style and color chosen by the interface designer. Several styles of text i/o objects are available from the Inputs, Textio, and GUI Models Libraries. These are design files **inputs.dsn, textio.dsn,** and **gui.dsn** or **guiwin.dsn** to be precise, found in the models sub-directory of the Altia software installation. They are organized for display within an Altia Models View (press the **Libraries** button in the Altia Design Editor to initiate the opening of a Models View). To learn specific details about text i/o objects, select the **Textio Models** option from the **Help** menu found in any Altia Models View.

AtSendText is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaSendText acts on the currently active Altia interface connection.

Designs created prior to software release 2.0 could only use the obsoleted (but still supported) text input and output objects instead of the newer and more versatile text i/o object. The obsoleted text input and output objects have a "putchar" animation instead of a "text" and "character" animation.

2.26.4 Return Values

If a communications failure occurs, these functions return -1. Otherwise, 0 is returned.

2.26.5 Windows Dependencies

To improve inter-process communications performance, events from AtSendText and altiaSendText calls are buffered on the application's side until a synchronous call such as AtNextEvent/altiaNextEvent or AtPollEvent/altiaPollEvent is performed or the program returns to AtMainLoop from a callback if AtMainLoop is being used. If this behavior produces undesirable effects, a call to

AtFlushOutput/altiaFlushOutput can be made to force buffered events to be sent immediately. Or, turn off buffering permanently with a call to AtCacheOutput/altiaCacheOutput.

2.26.6 See Also

AtSendEvent, altiaSendEvent, AtCacheOutput, AtFlushOutput, altiaCacheOutput, altiaFlushOutput

2.27 AtStartInterface, altiaStartInterface, AtStopInterface, altiaFetchArgcArgv

2.27.1 Name

AtStartInterface	Toolkit function to start an Altia interface and connect to it.
AtStopInterface	Toolkit function to stop an Altia interface and disconnect from it.
altiaStartInterface	Simplified version of AtStartInterface, but still a toolkit function.
altiaFetchArgcArgv	Utility function for Windows programs.

2.27.2 C Synopsis

```
AtConnectId AtStartInterface(const char *designFile, const char *rtmFile,
                             int editMode, int argc, const char *argv[]);

void AtStopInterface(AtConnectId connectId);
```

2.27.3 C Windows Synopsis

```
AtConnectId altiaStartInterface(const char *dsnFile, int editMode,
                                const char *cmdLine);

int altiaFetchArgcArgv(void *hInstance, char *lpCmdLIne, int argsize,
                       char **argv, int *argc);
```

2.27.4 Description

AtStartInterface will start a copy of the *Altia Runtime* interface executable, load it with the design file whose path is given by the string pointed to by *designFile*, configure it with application resource information from the Runtime Configuration File whose path is given by the string pointed to by *rtmFile*, and pass the executable additional parameters from the argument list described by *argc* and *argv*.

After the interface is started, a connection to it will be established and the id for this connection is the return value of AtStartInterface.

The *designFile* parameter must be non-NULL and point to a valid Altia interface design file or there must be an argument pair in the *argv* list of the form **-file** *FILENAME* where *FILENAME* is a valid Altia interface design file. If the file doesn't exist or isn't a valid design file, this function may fail in mysterious ways or a subsequent call to send or receive events from the interface will fail.

The *rtmFile* parameter may be NULL. If it is, then the *argv* list will be searched for an argument pair of the form **-defaults** *FILENAME* where *FILENAME* is the Runtime Configuration File. If no match is found in the *argv* list, then a Runtime Configuration File name will be constructed from the design file name by replacing the last suffix of the design file with .rtm or by simply adding .rtm if no suffix is found. A non-existent Runtime Configuration File will not cause a fatal error. The *Altia Runtime* interface will simply configure itself to a default specification.

If the *editMode* parameter is one (1), then the *Altia Design* Editor will be started instead of the *Altia Runtime*. In addition, the *Altia Design* Editor is started if the argument **-edit** is found in the *argv* list. If the Altia Design Editor is not found, an attempt will be made to open the Altia FacePlate Editor. If *editMode* is

two (2), an attempt is made to start Altia FacePlate. If FacePlate is not found, then an attempt will be made to open Altia Design.

If a program is linked with liblan.a on a UNIX system, AtStartInterface automatically constructs a unique file name for the domain socket to be used for the Altia interface connection. If an argument pair of the form **-port** *PORTNAME* is found in the *argv* list, then *PORTNAME* is used to construct the port name; however, the name may get customized in an attempt to guarantee its uniqueness in order to minimize conflicts with other possibly similar port names. After AtStartInterface successfully completes, the actual port name used can be retrieved with a call to AtGetPortName.

On Microsoft Windows systems, domain sockets are not supported. AtStartInterface will construct a network socket port number instead. Another alternative is to link with libdde.lib instead of liblan.lib to use a DDE service for client/interface communications. This is the only option on Windows systems not equipped with TCP/IP socket software or hardware. As on UNIX systems, an argument pair of the form **-port** *PORTNAME* in the *argv* list is used to specify a particular socket number or DDE name.

AtStartInterface also parses the *argv* list for the argument pair **-retry** *COUNT* to set the count for retrying a failed connection. If no retry count is found, the retry count is set to 0 which means an unlimited number of transmission failures are allowed. If desired, AtRetryCount can be called to set the retry count to something other than 0 after AtStartInterface completes. This is not recommended since the program is controlling the interface and it is best that the program has a chance to exit gracefully. That is, it should call AtStopInterface before it exits so that abandoned domain socket files get removed if any exist.

Finally, AtStartInterface also parses the *argv* list for the argument pair **-debug** *LEVEL* where *LEVEL* is one of the values 0, 1, or 2. This turns on different levels of debug. Level 0 is debug off. In level 1 debug, messages are sent to **stderr** each time a connection is opened or closed, a callback is added or removed, or an event is received. Level 2 debug includes level 1 messages and adds messages for each time the program sends an event to an Altia interface using AtSendEvent or AtSendText.

AtStartInterface actually attempts to fork and execute the *Altia Runtime* executable, altiart.out (UNIX) or altiart.exe (PC). If the executable is not in the current working directory and the ALTIAHOME environment variable is set, it uses it to construct a path to altiart.out of the form $ALTIAHOME/bin/altiart.out (%ALTIAHOME%\bin\altiart.exe on Windows). If the ALTIAHOME search fails, then the elements of the PATH environment variable are searched to find the executable. As a last resort, AtStartInterface looks for altiart.out in /usr/altia[fp]/bin (altiart.exe in \usr\altia[fp]\bin or \altia[fp]\bin on Windows). If the search is unsuccessful, AtStartInterface will return an error. This same search algorithm is applied to the Editor start-up program named **altia[fp]** if *editMode* is non-zero or **-edit** is found in the *argv* list.

Other arguments recognized by *Altia Runtime* or the Editor can be passed as entries in the *argv* list and they will be filtered to the Altia interface when it is started. Normally, the *argc* and *argv* parameters are the same parameters passed to the **main** subroutine of the C or C++ program. The *argc* parameter is a count of the number of valid entries in the *argv* list and *argv* is an array of pointers to strings holding the arguments.

An interface and connection started with AtStartInterface can be terminated with a call to AtStopInterface. AtStopInterface will close the connection and then attempt to kill the interface. If AtCloseConnection is called instead or the program just exits, the interface will be left running and only the connection will be closed. This might be useful if other programs are going to connect to the same Altia interface. The last program to exit can call AtStopInterface before calling exit. AtStopInterface terminates the interface by first sending an altiaQuit event to it. This will normally stop the interface unless the quit event name has been changed by a Altia*quitFunction: resource specification. If sending altiaQuit doesn't work, AtStopInterface tries to kill the interface with a signal. If the interface is an Editor session, AtStopInterface will not try to terminate it. The user must terminate the session manually.

altiaStartInterface is a simplified version of AtStartInterface for Windows programs. Instead of taking *argc* and *argv*, it takes a single string of space separated arguments.

altiaFetchArgcArgv is a utility for Windows users linking with libddew.lib and supplying their own WinMain function (libdde.lib supplies its own WinMain so that the application code only requires a standard main function). altiaFetchArgcArgv takes the command line that is passed to the application via the WinMain function and returns an argument array and count that can be used as the *argc* and *argv* arguments to AtStartInterface. *hInstance* is the first parameter from the WinMain function and is used to get the calling program's executable name. *lpCmdLine* is the command line parameter from WinMain. The caller must supply a character pointer array *argv* whose elements will be set by altiaFetchArgcArgv to point to the individual arguments of *lpCmdLine*. To avoid exceeding the array, the caller must supply the *argv* array size with *argsize*. The integer pointed to by *argc* will be set to the number of individual arguments found in *lpCmdLine*. The character strings pointed to by *argv* will be static data and should not be freed by the caller. Please note that altiaFetchArgcArgv will alter *lpCmdLine* by replacing spaces with null characters to construct the strings for the *argv* list. If *lpCmdLine* must be used again, a copy of it should be passed to altiaFetchArgcArgv, but the copy should have appropriate scope since elements of *argv* will point into it after altiaFetchArgcArgv returns.

2.27.5 Return Values

AtStartInterface and altiaStartInterface return a non-negative connection id if successful. If an Altia interface could not be started or a connection could not be established, -1 is returned.

AtStopInterface has no return value.

altiaFetchArgcArgv returns the actual number of arguments placed in the *argv* array. If this value and the integer pointed to by *argc* differ, then not all of the arguments from *lpCmdLine* could fit into the *argv* array.

2.27.6 Example

The following C main routine starts an *Altia Runtime* interface with the design file named demo.dsn. The Runtime Configuration File is not specified - a file name will be constructed from the design file name or from a **-defaults** *FILENAME* argument pair in the *argv* list if one exists. After starting the interface, an event is sent to it and then it is stopped.

```
#include "altia.h"
main(argc, argv)
```

```
int argc;
char *argv[];
{
    AtConnectId connectId;
    if ((connectId = AtStartInterface("demo.dsn", NULL, 0, argc, argv)) < 0)
    {
        fprintf(stderr, "AtStartInterface failed\n");
        exit(1);
    }
    AtSendEvent(connectId, "start", (AltiaEventType) 1);
    AtStopInterface(connectId);
}
```

2.27.7 See Also

AtOpenConnection, AtCloseConnection, AtRetryCount, altiaRetryCount, AtGetPortName

2.28 AtUnselectAllEvents, altiaUnselectAllEvents

2.28.1 Name

AtUnselectAllEvents	Toolkit function to unselect all events being received from an Altia interface.
altiaUnselectAllEvents	Base library version of AtUnselectAllEvents.

2.28.2 C Synopsis

```
int AtUnselectAllEvents(AtConnectId connectId);

int altiaUnselectAllEvents();
```

2.28.3 Description

Either function is called to discontinue the receiving of *all* Altia interface events. Unselecting events is only meaningful if events were previously selected with a call to AtSelectEvent/altiaSelectEvent or AtSelectAllEvents/altiaSelectAllEvents. If there are instances of events waiting to be processed, they will be destroyed. That is to say, all events will be removed from the queue of waiting events that is normally processed with calls to AtNextEvent/altiaNextEvent or AtMainLoop.

AtUnselectAllEvents is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaUnselectAllEvents acts on the currently active Altia interface connection.

Use of these functions is not usually necessary for the typical applications program. However, some applications find it necessary to temporarily discontinue receipt of all events - maybe because of a dramatic change in the state of the interface - and then continue receipt again at a later time.

2.28.4 Return Values

Zero (0) is returned upon successful completion. If a communications failure occurs, -1 is returned.

2.28.5 See Also

AtUnselectEvent, altiaUnselectEvent, AtSelectEvent, altiaSelectEvent, AtSelectAllEvents, altiaSelectAllEvents, AtNextEvent, altiaNextEvent, AtPending, altiaPending

2.29 AtUnselectEvent, altiaUnselectEvent

2.29.1 Name

AtUnselectEvent	Toolkit function to unselect to receive a particular type of Altia interface event.
altiaUnselectEvent	Base library version of AtUnselectEvent.

2.29.2 C Synopsis

```
int AtUnselectEvent(AtConnectId connectId, const char *eventName);

int altiaUnselectEvent(const char *eventName);
```

2.29.3 Description

Either function is called to discontinue the receiving of a specific Altia interface event type. The *eventName* parameter gives the name of the events that should no longer be received. Unselecting an event is only meaningful if the event was previously selected with a call to AtSelectEvent/altiaSelectEvent or AtSelectAllEvents/altiaSelectAllEvents. If there are instances of the event waiting to be processed, they will be destroyed. That is to say, they will be removed from the queue of waiting events that is normally processed with calls to AtNextEvent/altiaNextEvent or AtMainLoop.

AtUnselectEvent is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaUnselectEvent acts on the currently active Altia interface connection.

Use of these functions is not usually necessary for the typical applications program. However, some applications find it necessary to temporarily discontinue receipt of a particular event or events - maybe because of a dramatic change in the state of the interface - and then continue receipt again at a later time.

2.29.4 Return Values

Zero (0) is returned upon successful completion. If a communications failure occurs, -1 is returned.

2.29.5 See Also

AtSelectEvent, altiaSelectEvent, AtSelectAllEvents, altiaSelectAllEvents

2.30 altiaClearConnect

2.30.1 Name

altiaClearConnect	Base library function to clear internal data structures after a disconnect.

2.30.2 C Synopsis

```
void altiaClearConnect();
```

2.30.3 Description

This function has no toolkit counterpart. It is actually intended for use solely by the internals of the toolkit and its functionality could change in the future. This function clears the internal data structures associated with an Altia interface connection if the connection has already been closed with a call to altiaDisconnect. If a call to altiaRemoveConnect is going to be made, then it must precede the call to altiaClearConnect.

2.30.4 Return Values

None

2.31 altiaConnect, altiaDisconnect, altiaStopInterface

2.31.1 Name

altiaConnect	Base library function to connect an application program to an Altia interface.
altiaDisconnect	Base library function to disconnect an application program from an Altia interface.
altiaStopInterface	Base library function to disconnect from and terminate an Altia interface.

2.31.2 C Synopsis

```
int altiaConnect(const char *portName);

void altiaDisconnect();

void altiaStopInterface();
```

2.31.3 Description

altiaConnect opens a connection to an Altia interface. It is used exclusively when only base library functions are being used. AtOpenConnection should be called to open a connection when toolkit routines are being used - this is the preferred method.

A call to altiaConnect is not normally necessary since the other base library routines will attempt to open a default connection if one is not already opened. However, if it is desirable to open a connection based on a unique name, altiaConnect must be called with the *portName* parameter pointing to a string containing the desired name for the connection. Assigning unique names to instances of Altia interfaces allows multiple Altia sessions to run on a single workstation. Passing NULL for *portName* opens the default connection if it is not already opened. Using the default connection only allows one Altia interface session on a computer.

On UNIX systems, a domain or network socket serves as the connection facility between an Altia interface and one or more application programs (which are sometimes referred to as clients). On Microsoft Windows, only network sockets or DDEs are available. An application program must be linked with liblan.a on UNIX or liblan.lib on the PC to use sockets as the connection facility. To use DDEs on the PC, an application program must be linked with libdde.lib.

UNIX domain sockets allow communication between processes residing on the same workstation. A domain socket is specified by using a filename syntax for the string pointed to by *portName* (e.g., "/tmp/screen"). A full path for the file name is recommended, but not necessary. When *portName* is non-NULL, the Altia interface that the client is connecting to must be started with the command line arguments **-port** *PORTNAME* where *PORTNAME* is an exact match for the string referenced by *portName*. If *portName* is NULL, a default domain socket is used and no special parameter is required when starting the Altia interface. The default domain socket name is created in the /usr/tmp directory with the name **vSe.***HOSTNAME* where *HOSTNAME* is the name of the local machine.

Network sockets are used for communicating between processes on the same computer or on networked computers. Altia supports the execution of an Altia interface on one machine and one or more clients on networked machines when a network socket is used as the communications medium. A network socket is specified by passing a string to altiaConnect that follows the syntax of *HOSTNAME*:*SOCKET* where *SOCKET* may be a number or it may be an alias name for a socket number as specified in the /etc/services or equivalent file. If *HOSTNAME* is omitted, than the name of the current host is assumed. The Altia interface that the client is connecting to must be started with the command line argument **-port** *PORTNAME* where *PORTNAME* is an exact match for the string passed to altiaConnect. On Windows 3.1 and 9x systems, a **-lan** argument must accompany **-port** *PORTNAME* to enable socket initialization.

For Windows systems that do not have TCP/IP socket software or hardware, a DDE style connection is available. The Altia interface and its client program(s) must reside on the same computer - networked DDE service is not supported - and programs must be linked with the libdde.lib program library instead of liblan.lib. If *portName* is NULL, then a default DDE Service (AltDDE) will be opened. The Altia interface will open the same service by default as well. If *portName* is non-NULL, then the Altia interface must be started with the command line argument **-dde** *DDENAME* where *DDENAME* is an exact match for the *portName* string passed to altiaConnect. A DDE Service Name can be a logical name of your choosing.

Each base library call attempts to establish or reestablish a connection if contact is lost. If a call cannot reestablish a connection or looses connection in the middle of a transaction, it returns -1. Under this mode of operation, it is up to the code making calls to the library to detect when a connection has gone down (by testing return values from the library calls) and decide if it should exit under such circumstances. Exiting is probably a good idea under normal operating conditions. As an alternative approach, a call to altiaRetryCount can be made to change the infinite retry count to something finite. If the retry count is *N*, then the library will call exit when the *N*-th call to a library function fails. Altia uses a retry count of 1 for all of its demos. This insures that an application program isn't left running in background after the Altia interface has been closed by the customer.

Note that altiaConnect can only succeed if there is an Altia interface already running that is *serving* the selected domain or network socket or DDE service.

altiaDisconnect closes the connection to an Altia interface if one is currently opened. It is not usually necessary to close a connection since it automatically closes when contact is lost.

altiaStopInterface attempts to terminate the Altia interface process by sending an altiaQuit event to the interface and then it closes the connection to the interface. If the interface is configured in its standard mode, it will receive this event and terminate itself. If the interface process is an Editor session, then it will ignore the message to insure against accidental loss of design changes. Please note that if AtStartInterface was originally called to start the interface, then its companion toolkit function AtStopInterface should be called to terminate it.

2.31.4 Return Values

altiaConnect returns 0 upon successful completion. If no connection is established, -1 is returned and errno (the global error value variable) is set to reflect the cause of the error. The altiaDisconnect and altiaStopInterface routines have no return values.

2.31.5 Examples

The following UNIX code segment attempts to open a domain socket connection to an Altia interface using the file name /users/guest/altia1 as the base name for the socket:

```
if (altiaConnect("/users/guest/altia1") < 0)
{
    fprintf(stderr "Cannot connect to Altia interface - exiting\n");
    exit(1);
}
```

The next code segment attempts to open a network socket connection to an Altia interface running on machine **lab1**. The socket number is specified explicitly as **12345**. This code sets the retry count to infinity (actually this is unnecessary since the default is infinity, but it is good coding practice) and waits in an infinite loop for the connect to succeed. On machine **lab1**, the Altia interface needs to be started with the arguments **-port lab1:12345** or just **-port :12345** (the **lab1** portion is redundant since the machine must be **lab1**). On Windows 3.1 and 9x, the additional argument **-lan** must be included to enable socket initialization.

```
altiaRetryCount(0);
while (altiaConnect("lab1:12345") < 0)
{
    sleep(1);
}
```

2.31.6 See Also

AtOpenConnection, AtCloseConnection, AtStartInterface, altiaStartInterface, AtStopInterface, altiaFetchArgcArgv

2.32 altiaRemoveConnect

2.32.1 Name

altiaRemoveConnect	Base library function to remove domain socket or pipe files after a disconnect.

2.32.2 C Synopsis

```
void altiaRemoveConnect();
```

2.32.3 Description

This function has no toolkit counterpart. It is actually intended for use solely by the internals of the toolkit and its functionality could change in the future. This function removes the domain socket file or pipe files associated with an Altia interface connection if the connection has already been closed with a call to altiaDisconnect. If a call to altiaClearConnect is going to be made, then the call to altiaRemoveConnect must precede it.

2.32.4 Return Values

None

2.33 altiaSelectConnect, altiaGetConnect

2.33.1 Name

altiaSelectConnect	Base library function to change the currently selected Altia interface connection.
altiaGetConnect	Base library function to get the currently selected Altia interface connection.

2.33.2 C Synopsis

```
void altiaSelectConnect(int connectNum);

int altiaGetConnect();
```

2.33.3 Description

This version of the Altia base library supports multiple Altia interface connections from a single application program. It does this without changing the usage for the simple single connection approach. To establish and use multiple connections, a connection index must be assigned to each connection. The connection index for the first connection would be 0. This is also the default connection index for the case where only 1 connection is established. The second connection would be assigned index 1, and so on, up to index 9. That is, the library supports 10 simultaneous connections at a time.

All library calls, including altiaConnect and altiaDisconnect, operate on the currently selected connection. A connection is selected with a call to altiaSelectConnect, passing the index for the connection you wish to select via the *connectNum* parameter. Until another call to altiaSelectConnect, all library calls will act on the selected connection.

If a connection index less than 0 or greater than 9 is passed to altiaSelectConnect, it is ignored and the previously selected connection remains selected.

To query for the current connection index, call altiaGetConnect. Its return value is the index.

The need to use altiaSelectConnect and altiaGetConnect is obsoleted by the toolkit library's support for multiple connections. All toolkit routines accept a connection id as a parameter where applicable. It is recommended that the toolkit function AtOpenConnection be used to establish multiple connections for a single application program.

2.33.4 Examples

The following example opens domain socket connections to 2 different interfaces and sends events to each of them independently.

```
altiaSelectConnect(0);
if (altiaConnect("/users/guest/altia1") < 0)
    exit(1);

altiaSelectConnect(1);
if (altiaConnect("/users/guest/altia2") < 0)
    exit(1);
```

```
altiaSelectConnect(0);
altiaSendEvent("start", 1);

altiaSelectConnect(1);
altiaSendEvent("start", 1);
```

2.33.5 Return Values

altiaSelectConnect has no return value. altiaGetConnect returns the index of the currently selected connection or 0 by default.

2.33.6 See Also

AtOpenConnection, AtCloseConnection, AtStartInterface, altiaStartInterface, AtStopInterface, altiaFetchArgcArgv, altiaConnect, altiaDisconnect, altiaStopInterface

2.34 altiaSleep

2.34.1 Name

altiaSleep	Base library function to provide a system-independent millisecond sleep capability.

2.34.2 C Synopsis

```
void altiaSleep(unsigned long milliseconds)
```

2.34.3 Description

This function has no toolkit counterpart. It provides a system-independent method for specifying a sleep period for an application program. The time to sleep is given in milliseconds. The program will actually suspend its execution for the specified period. In multi-tasking environments such as UNIX, Windows/NT, Windows *95*,and Windows 3.1, there is no guarantee as to the accuracy of the actual interval; however, it is typically accurate enough for delays relating to human response and perception. Requesting a sleep for 10's of milliseconds will probably not yield a very accurate result; however, requests for delays of 100 or more milliseconds can typically be handled with acceptable accuracy on most computer systems.

This function is completely independent of any Altia interface connection. It may be called at any time by the application whether or not an Altia interface connection is opened and active.

Very Important Windows 16-bit Dependencies

The "L" notation on a constant (for example, **altiaSleep(1000L);**) must be used when specifying *milliseconds* as a number. Failure to do so will send the program into outer cyberspace when it executes because altiaSleep is expecting an unsigned long value and not a 16-bit integer.

2.34.4 Return Values

None

2.35 altiaSuppressErrors

2.35.1 Name

altiaSupressErrors	Base library function to suppress error messages during altiaConnect calls.

2.35.2 C Synopsis

```
void altiaSuppressErrors(int yes);
```

2.35.3 Description

This function has no toolkit counterpart and its functionality could change in the future. If the *yes* parameter is non-zero, this function suppresses **stderr** error messages during an Altia interface connect or reconnect. Passing a *yes* parameter of 0 turns error messages on again.

This function **must** be called when using the **`liblan`** library with a Windows application or else the error messages sent to **stderr** will cause the application to crash. Moreover, the `extern int _AtNoConnectErrors` must be declared and the call to `altiaSuppressErrors` must set its value when it is called.

2.35.4 Example

The following C main routine turns off error messages to **stderr** before starting an *Altia Runtime* interface with the design file named demo.dsn. After starting the interface, error messages are turned back on, an event is sent to Altia, and then it is stopped.

```
#include "altia.h"
extern int _AtNoConnectErrors;
main(int argc, char *argv[])
{
    AtConnectId connectId;
    altiaSuppressErrors(_AtNoConnectErrors = 1);
    if ((connectId = AtStartInterface("demo.dsn", NULL, 0, argc, argv)) < 0)
    {
        fprintf(stderr, "AtStartInterface failed\n");
        exit(1);
    }
    altiaSuppressErrors(_AtNoConnectErrors = 0);
    AtSendEvent(connectId, "start", (AltiaEventType) 1);
    AtStopInterface(connectId);
}
```

2.35.5 Return Values

None

3 Routines for Opening and Manipulating Views and Designs

3.1 Introduction

This section describes Altia library and toolkit routines that allow for the programmatic creation and manipulation of Altia interface views. Altia design open and manipulation routines allow applications to open and close design files for a single Altia interface session. When a new design is opened, one or more views into the design can be opened using the view open and manipulation routines. More than one design can be open at a time with different views displaying different designs. Closing a design closes the views associated with the design as well.

Altia interface views are windows that display all or a portion of a design. A design can extend beyond the boundary of a view. A view can be thought of as a portal into the design *universe*. Some designs fit entirely into the area of a view. In these situations, a single static view is probably sufficient and this is what the Altia Graphics Editor and Altia Runtime provide by default. Other designs may actually be very large. In this case, it may be desirable to dynamically create views to show different portions of the design at different times. The routines in this chapter of the Altia API Reference manual are used to programmatically create and manipulate views, or to actually change the entire design that is displayed in a new or existing view.

When an application is connected to an Altia interface that is being displayed in the Altia Graphics Editor, a programmatic request to open a new view creates a view identical in layout to a view opened with the **New** option in the **View** menu. The view will have a Banner, Menu Bar, and Panner. Because of these additions, the new view's size may not be the size requested by the program. If a new design file is opened into the Graphics Editor programmatically, it will always replace the current design. The Graphics Editor does not support the opening of more than one design at a time. In the Altia Runtime environment, these constraints are removed. A view will only show the design and no Banner, Menu Bar, or Panner. This allows the view to size and resize exactly as requested by the application. And, different views can display different design files simultaneously in the runtime environment.

To provide complete backwards compatibility, the view and design open and manipulation routines are provided in base library and toolkit form. Toolkit versions begin with the *At* prefix and base library versions use an *altia* prefix. Programs using base library and toolkit routines must link with the UNIX liblan.a object library found in the lib or libfloat directory of the Altia software installation. On systems running Microsoft Windows, a program is typically linked with libdde.lib, or it can be linked with liblan.lib if TCP/IP socket software and hardware is available. On all types of systems, the include file for the library is named altia.h and it resides in the same directory as the object libraries.

The Altia Design software installation provides a **demos** directory containing various complete Altia interface examples. The files found in the **views** sub-directory provide a working example of view creation and manipulation from an application program. See the **README** file in the **views** directory for more details. As a quick summary, here is a table of the view and design manipulation routines:

Routine	Description
AtOpenDesignFile	Opens a new design into an existing Altia interface session and assigns it an id.
AtSetDesignId	Sets the active design id for opening or closing views into the design.
AtCloseDesignId	Closes the design identified by a specific design id.
AtOpenSimpleView	Opens a new view using default size, placement, orientation, magnification.
AtOpenView	Opens a new view and caller chooses initial size, orientation, magnification.
AtOpenViewPlaced	Opens a new view as with AtOpenView, but places it in windowing system
AtOpenWindowView	Opens a new view into an existing Windowing System window.
AtMoveView	Changes the area of the design that is shown in a view.
AtSizeView	Changes the size of a view.
AtMagnifyView	Changes the magnification factor of a view.
AtGetViewSize	Gets the width and height for a specific view.
AtGetViewWindowId	Gets the Windowing System's window id number for a specific view.
AtCloseView	Closes a view.
altiaOpenDesignFile	Base Library version of AtOpenDesignFile.
altiaSetDesignId	Base Library version of AtSetDesignId.
altiaCloseDesignId	Base Library version of AtCloseDesignId.
altiaOpenSimpleView	Base Library version of AtOpenSimpleView.
altiaOpenView	Base Library version of AtOpenView.
altiaOpenViewPlaced	Base Library version of AtOpenViewPlaced.
altiaOpenWindowView	Base Library version of AtOpenWindowView.
altiaMoveView	Base Library version of AtMoveView.

Routine	Description
altiaSizeView	Base Library version of AtSizeView.
altiaMagnifyView	Base Library version of AtMagnifyView.
altiaGetViewSize	Base Library version of AtGetViewSize.
altiaGetViewWindowId	Base Library version of AtGetViewWindowId.
altiaCloseView	Base Library version of AtCloseView.

3.2 AtCloseDesignId, altiaCloseDesignId

3.2.1 Name

AtCloseDesignId	Toolkit function to close a design file opened into an Altia interface session.
altiaCloseDesignId	Base library version of AtCloseDesignId.

3.2.2 C Synopsis

```
int AtCloseDesignId(AtConnectId connectId, int designId);

int altiaCloseDesignId(int designId);
```

3.2.3 Description

Both functions close the design file associated with *designId*. All views opened for the design are closed. The currently active design id is also set to the closed design id which affects future view open requests (see AtSetDesignId, altiaSetDesignId).

A program cannot close *designId* = 0 because it is the original design. To close *designId* = 0, a program must send an altiaQuit event (e.g., **AtSendEvent(connectId, "altiaQuit", (AltiaEventType) 1);**) to the Altia interface process or call AtStopInterface or altiaStopInterface to terminate the Altia interface process.

AtCloseDesignId is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaCloseDesignId acts on the currently active Altia interface connection.

3.2.4 Return Values

If a communications failure occurs, these functions return -1. If an illegal parameter is detected a value of -2 is returned. Otherwise, 0 is returned.

3.2.5 See Also

AtOpenDesignFile, altiaOpenDesignFile, AtSetDesignId, altiaSetDesignId, AtCloseView, altiaCloseView

3.3 AtCloseView, altiaCloseView

3.3.1 Name

AtCloseView	Toolkit function to close a view.
altiaCloseView	Base library version of AtCloseView.

3.3.2 C Synopsis

```
int AtCloseView(AtConnectId connectId, int viewId);

int altiaCloseView(int viewId);
```

3.3.3 Description

Both functions send a request to the Altia interface to close a view that was previously opened with AtOpenSimpleView/altiaOpenSimpleView, AtOpenView/altiaOpenView, or AtOpenViewPlaced/altiaOpenViewPlaced. If the view was opened into an existing window with AtOpenWindowView/altiaOpenWindowView, the window is left open, but it is no longer updated by the Altia interface process.

The identification number of the view to close is passed in *viewId*. Once a view is closed, its identification number may be reused.

A user can normally close any view manually using the **Close** or **Quit** option in the view's window banner menu. To suppress this feature, an application program can select to receive the special event "altiaCloseViewPending" with AtSelectEvent/altiaSelectEvent or register a callback for it with AtAddCallback. If this is done, the Altia interface process will not automatically close a view when the user chooses the **Close** or **Quit** option. Instead, an event with the name "altiaCloseViewPending" will be sent to the program and the value of the event will be the view number that the user is attempting to close (an event value of 0 would indicate the close is on the Main View). The program can then decide how to respond to the user's view close request.

AtCloseView is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaCloseView acts on the currently active Altia interface connection.

3.3.4 Return Values

If a communications failure occurs, these functions return -1. If *viewId* is an illegal value, -2 is returned. Otherwise, 0 is returned. Not all illegal id values can be detected prior to sending the close request to the Altia interface. The Altia interface does not return an error to the program if it receives an illegal close request. It simply will not close the view (although it is most likely no such view even exists).

3.3.5 See Also

AtOpenSimpleView, altiaOpenSimpleView, AtOpenView, altiaOpenView, AtOpenViewPlaced, altiaOpenViewPlaced, AtOpenWindowView, altiaOpenWindowView

3.4 AtGetViewSize, altiaGetViewSize

3.4.1 Name

AtGetViewSize	Toolkit function to get the width and height for a specific view.
altiaGetViewSize	Base library version of AtGetViewSize.

3.4.2 C Synopsis

```
int AtGetViewSize(AtConnectId connectId, int viewId, int *widthOut,
                  int *heightOut);

int altiaGetViewSize(int viewId, int *widthOut, int *heightOut);
```

3.4.3 Availability

These functions are only available in release 2.0 or later on UNIX workstations. On Windows systems, these functions are not available in release 2.0, but will be available in all releases following release 2.0.

3.4.4 Description

Both functions send a request to the Altia interface to get the width and height, in screen pixels, for the view with the view id number *viewId*. To get the width and height for the initial Main View, a value of 0 is passed in *viewId*. The caller must provide two pointers, *widthOut* and *heightOut*, to two integer variables. The functions return the current width and height settings for the view in the integer variables pointed to by *widthOut* and *heightOut*.

When a view is initially created or later resized by the user, a special event named "altiaViewResize" is generated by the Altia interface. The value of the event is the id number of the view that was created or resized. An application program can choose to select to receive "altiaViewResize" events with AtSelectEvent/altiaSelectEvent or register a callback for them with AtAddCallback. The program can then call AtGetViewSize or altiaGetViewSize to get the new size for the view.

AtGetViewSize is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaGetViewSize acts on the currently active Altia interface connection.

3.4.5 Return Values

If a communications failure occurs, these functions return -1. If *viewId* is an illegal value or *widthOut* or *heightOut* is a nil pointer, -2 is returned. Otherwise, 0 is returned. Not all illegal id values can be detected prior to sending the size request to the Altia interface. The Altia interface does not return an error to the program if it receives an illegal size request. It will simply return incorrect values in the integer variables pointed to by *widthOut* and *heightOut*.

3.4.6 Example

The simple program example on the following page selects to receive "altiaViewResize" events through a callback function. In the callback function, AtGetViewSize is called to get the new size for the view.

```
#include <stdio.h>
#include "altia.h"

void resizeHandler();

main(argc, argv)
int argc;
char *argv[];
{
    AtConnectId connectId;
    if ((connectId = AtOpenConnection(NULL, NULL, argc, argv)) < 0)
    {
        fprintf(stderr, "AtOpenConnection failed\n");
        exit(1);
    }
    AtAddCallback(connectId, "altiaViewResize", resizeHandler, NULL);
    AtMainLoop();
}

void resizeHandler(connectId, eventName, eventValue, data)
AtConnectId connectId;
char *eventName;
AltiaEventType eventValue;
AtPointer data;
{
    int width, height;
    AtGetViewSize(connectId, (int) eventValue, &width, &height);
    fprintf(stderr, "View %d resized to %d width, %d height\n",
           (int) eventValue, width, height);
}
```

3.4.7 See Also

AtGetViewWindowId, altiaGetViewWindowId

3.5 AtGetViewWindowId, altiaGetViewWindowId

3.5.1 Name

AtGetViewWindowId	Toolkit function to get the Windowing System's window id for a specific view.
altiaGetViewWindowId	Base library version of AtGetViewWindowId.

3.5.2 C Synopsis

```
int AtGetViewWindowId(AtConnectId connectId, int viewId, int *windowIdOut);

int altiaGetViewWindowId(int viewId, int *windowIdOut);
```

3.5.3 Availability

These functions are only available in release 2.0 or later on UNIX workstations. On Windows systems, these functions are not available in release 2.0, but will be available in all releases following release 2.0.

3.5.4 Description

Both functions send a request to the Altia interface to get the native Windowing System's window id number for the view with the id number *viewId*. To get the window id for the initial Main View, a value of 0 is passed in *viewId*. The caller must provide a pointer, *windowIdOut*, to an integer variable. The functions return the window id in the variable pointed to by *windowIdOut*.

AtGetViewWindowId is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaGetViewWindowId acts on the currently active Altia interface connection.

3.5.5 Disclaimer

The Windowing System window id for a view can be used to do Windowing System specific operations on the view's window.

NOTE: These types of operations may not be portable across different Windowing Systems or different hardware platforms. **The program developer takes on full responsibility for maintaining and supporting all Windowing System specific code. Altia may change the way views interface with specific Windowing Systems and is in no way responsible for the failure of Windowing System specific code as a result of these changes.**

3.5.6 Return Values

If a communications failure occurs, these functions return -1. If *viewId* is an illegal value or *windowIdOut* is a nil pointer, -2 is returned. Otherwise, 0 is returned. Not all illegal id values can be detected prior to sending the request for the window id to the Altia interface. The Altia interface does not return an error to the program if it receives an illegal view id. It will simply return an incorrect value in the integer variable pointed to by *windowIdOut*.

3.5.7 See Also

AtGetViewSize, altiaGetViewSize

3.6 AtMagnifyView, altiaMagnifyView

3.6.1 Name

AtMagnifyView	Toolkit function to change the magnification factor of objects in a view.
altiaMagnifyView	Base library version of AtMagnifyView.

3.6.2 C Synopsis

```
int AtMagnifyView(AtConnectId connectId, int viewId, double magnification);

int altiaMagnifyView(int viewId, double magnification);
```

3.6.3 Description

Both functions send a request to the Altia interface to change the magnification factor of Altia interface objects shown in a view previously opened with:

- **AtOpenSimpleView/altiaOpenSimpleView**
- **AtOpenView/altiaOpenView**
- **AtOpenViewPlaced/altiaOpenViewPlaced**
- **AtOpenWindowView/altiaOpenWindowView**

The identification number of the view to change is passed in *viewId*. Use a *viewId* of 0 to affect the Graphics Editor's Main View.

The *magnification* parameter specifies the view's new desired magnification factor. The factor must be a positive value greater than or equal to .01. As examples, a factor of .5 would cause objects in the view to be shown at one-half their normal size. With a factor of 2.0, objects in a view would appear twice their normal size.

The Altia interface may impose limits on the magnification range (it currently limits it to 1/16th (.0625) through 16 times normal size). If a request outside of this range is received, it will be truncated to the minimum or maximum of the range, whichever is appropriate.

AtMagnifyView is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaMagnifyView acts on the currently active Altia interface connection.

3.6.4 Return Values

If a communications failure occurs, these functions return -1. If *magnification* is less than .01, -2 is returned. Otherwise, 0 is returned. An illegal view id cannot be detected prior to sending the magnification request to the Altia interface. In such cases, the return value will be 0 and the magnification operation will simply not happen.

3.7 AtMoveView, altiaMoveView

3.7.1 Name

AtMoveView	Toolkit function to change the area of a design that is being shown in a view.
altiaMoveView	Base library version of AtMoveView.

3.7.2 C Synopsis

```
int AtMoveView(AtConnectId connectId, int viewId, int universeX, int universeY);

int altiaMoveView(int viewId, int universeX, int universeY);
```

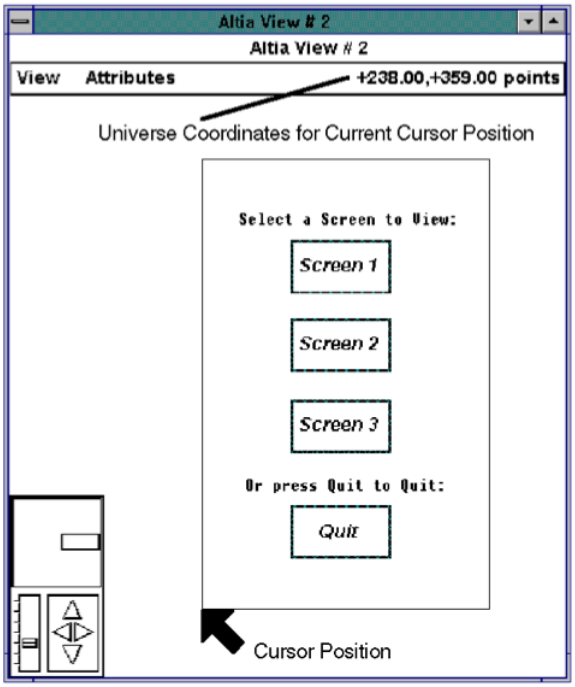
3.7.3 Description

Both functions send a request to the Altia interface to change the orientation of the design being shown within a view previously opened with:

- **AtOpenSimpleView/altiaOpenSimpleView**
- **AtOpenView/altiaOpenView**
- **AtOpenViewPlaced/altiaOpenViewPlaced**
- **AtOpenWindowView/altiaOpenWindowView**

The identification number of the view to change is passed in *viewId*. You can change the orientation of the design in the default Main View by passing a *viewId* of 0.

The *universeX* and *universeY* parameters specify the design universe coordinates that are to appear in the lower left corner of the view. Universe coordinates are actively displayed by the Altia Design Editor interface. If the Altia Design Editor is in edit mode (versus run mode), an x,y coordinate pair appears on the right hand side of each view's menu banner as the cursor is moved around the view. This information can be used to determine *universeX* and *universeY* parameters for AtMoveView or altiaMoveView calls. For instance, if you wish to position an instrument front panel in a view with the lower left corner of the panel in the lower left corner of the view, simply record the x,y pair shown in the View Menu Bar when the cursor is position on the lower left corner of the panel. Use the values of the x,y pair for the *universeX* and *universeY* parameters. See the figure below for an illustration.



AtMoveView is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaMoveView acts on the currently active Altia interface connection.

3.7.4 Return Values

If a communications failure occurs, these functions return -1. Otherwise, 0 is returned. An illegal view id or inappropriate universe x or y value cannot be detected prior to sending the move request to the Altia interface. In such cases, the return value will be 0 and the move operation will simply not happen.

3.8 AtOpenDesignFile, altiaOpenDesignFile

3.8.1 Name

AtOpenDesignFile	Toolkit function to open a new design into an existing Altia interface session.
altiaOpenDesignFile	Base library version of AtOpenDesignFile.

3.8.2 C Synopsis

```
int AtOpenDesignFile(AtConnectId connectId, int designId, const char *fileName);

int altiaOpenDesignFile(int designId, const char *fileName);
```

3.8.3 Description

Both functions open a new Altia design file named *fileName* and assign the id passed in *designId* to the new design. The currently active design id is also set to the new design id which affects future view open requests (see AtSetDesignId, altiaSetDesignId). If designId is an id of an already opened design file, then that design gets replaced by the new design. A *designId* of zero (0) will change the original design that was opened. If *designId* is not associated with any existing open design file, then a new design is opened. The various view open functions must then be called to open one or more views into the new design. A design and its views can be closed with a call to AtCloseDesignId.

AtOpenDesignFile is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaOpenDesignFile acts on the currently active Altia interface connection.

3.8.4 Return Values

If a communications failure occurs, these functions return -1. If an illegal parameter is detected, -2 is returned. Otherwise, 0 is returned. A non-existent or invalid file error is not detected by these functions. Altia will just ignore the request but the return value from the functions will be 0.

3.8.5 See Also

AtSetDesignId, altiaSetDesignId, AtCloseDesignId, altiaCloseDesignId, AtOpenView, altiaOpenView, AtOpenViewPlaced, altiaOpenViewPlaced, AtOpenWindowView, altiaOpenWindowView, AtCloseView, altiaCloseView

3.9 AtOpenSimpleView, altiaOpenSimpleView

3.9.1 Name

AtOpenSimpleView	Toolkit function to open a new view with minimal customization.
altiaOpenSimpleView	Base library version of AtOpenSimpleView.

3.9.2 C Synopsis

```
int AtOpenSimpleView(AtConnectId connectId, int newViewId,
                     const char *viewName);

int altiaOpenSimpleView(int newViewId, const char *viewName);
```

3.9.3 Description

Both functions send a request to the Altia interface to open a new view. The size, position of objects within the view and magnification factor of the view are determined by the Altia interface from specifications in the Runtime Configuration File (*.rtm* file), *X* resource database, *X* defaults resource file, or Altia application defaults file (with precedence as listed). Position of the view within the Windowing System is determined by the Windowing System's configuration.

The identification number for the view is determined by the caller and passed as the *newViewId* parameter. The new id must be unique with respect to all existing views which have been created programmatically. For example, if a view was already opened with a *newViewId* of 1, then a second view could be opened with a *newViewId* of 2 or greater. View id numbers must be greater than or equal to 1. A view id may be reused after a view has been closed with AtCloseView or altiaCloseView or closed by the user from a Window Manager menu option.

The *viewName* parameter provides a method for specifying a name for the new view. The name will appear in the Window Manager banner portion of the view. The *viewName* is typically entered as a double-quoted string such as “Extra View” or NULL for a default name.

AtOpenSimpleView is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaOpenSimpleView acts on the currently active Altia interface connection.

3.9.4 Return Values

If a communications failure occurs, these functions return -1. If an illegal parameter is detected, -2 is returned. Otherwise, 0 is returned. Not all illegal parameters can be detected prior to sending the open request to the Altia interface. The Altia interface does not return an error to the program if it receives an illegal open request. It simply will not open the new view.

3.9.5 See Also

AtOpenView, altiaOpenView, AtOpenViewPlaced, altiaOpenViewPlaced, AtOpenWindowView, altiaOpenWindowView, AtCloseView, altiaCloseView

3.10 AtOpenView, altiaOpenView

3.10.1 Name

AtOpenView	Toolkit function to open a new view with customizations.
altiaOpenView	Base library version of AtOpenView.

3.10.2 C Synopsis

```
int AtOpenView(AtConnectId connectId, int newViewId, const char *viewName,
               int universeX, int universeY, int width, int height,
               double magnification);

int altiaOpenView(int newViewId, const char *viewName, int universeX,
                  int universeY, int width, int height, double magnification);
```

3.10.3 Description

Both functions send a request to the Altia interface to open a new view. The orientation of the design within the view, size of the view window, and magnification factor of the view can all be specified. The position of the view within the Windowing System is determined by the Windowing System's configuration.

The identification number for the view is determined by the caller and passed as the *newViewId* parameter. The new id must be unique with respect to all existing views which have been created programmatically. For example, if a view was already opened with a *newViewId* of 1, then a second view could be opened with a *newViewId* of 2 or greater. View id numbers must be greater than or equal to 1. A view id may be reused after a view has been closed with AtCloseView or altiaCloseView or closed by the user from a Window Manager menu option.

The *viewName* parameter provides a method for specifying a name for the new view. The name will appear in the Window Manager banner portion of the view. The *viewName* is typically entered as a double-quoted string such as "Extra View" or NULL for a default name.

The caller may specify the area of the design that should appear within the view. The parameters *universeX* and *universeY* specify the design universe coordinates that are to appear in the lower left corner of the new view. Please see AtMoveView, altiaMoveView for more details on determining design universe coordinates. The width and height of the view window as well as the magnification factor that should be applied to the design universe may also be specified through the *width*, *height*, and *magnification* parameters, respectively. Lower left corner coordinates, size, or magnification of the view may be left up to the Altia interface system by using the constants **DefaultViewX**, **DefaultViewY**, **DefaultViewHeight**, **DefaultViewWidth**, or **DefaultViewMag** for *universeX*, *universeY*, *height, width* and *magnification*, respectively.

AtOpenView is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaOpenView acts on the currently active Altia interface connection.

3.10.4 Return Values

If a communications failure occurs, these functions return -1. If an illegal parameter is detected, -2 is returned. Otherwise, 0 is returned. Not all illegal parameters can be detected prior to sending the open request to the Altia interface. The Altia interface does not return an error to the program if it receives an illegal open request. It simply will not open the new view.

3.10.5 See Also

AtOpenSimpleView, altiaOpenSimpleView, AtOpenViewPlaced, altiaOpenViewPlaced , AtOpenWindowView, altiaOpenWindowView, AtCloseView, altiaCloseView

3.11 AtOpenViewPlaced, altiaOpenViewPlaced

3.11.1 Name

AtOpenViewPlaced	Toolkit function to open a new view with customizations and placement.
altiaOpenViewPlaced	Base library version of AtOpenViewPlaced.

3.11.2 C Synopsis

```
int AtOpenViewPlaced(AtConnectId connectId, int newViewId, const char *viewName,
                     int referenceViewId, int referenceX, int referenceY,
                     int universeX, int universeY, int width, int height,
                     double magnification);

int altiaOpenViewPlaced(int newViewId, const char *viewName, int referenceViewId,
                        int referenceX, int referenceY, int universeX,
                        int universeY, int width, int height,
                        double magnification);
```

3.11.3 Description

Both functions send a request to the Altia interface to open a new view. The placement of the view within the Windowing System may be specified as well as the position of objects within the view, size of the view window, and magnification factor of the view.

The identification number for the view is determined by the caller and passed as the *newViewId* parameter. The new id must be unique with respect to all existing views which have been created programmatically. For example, if a view was already opened with a *newViewId* of 1, then a second view could be opened with a *newViewId* of 2 or greater. View id numbers must be greater than or equal to 1. A view id may be reused after a view has been closed with AtCloseView or altiaCloseView or if the view was closed by the user from a Window Manager menu option.

The *viewName* parameter provides a method for specifying a name for the new view. The name will appear in the Window Manager banner portion of the view The *viewName* is typically entered as a double-quoted string such as "Extra View" or NULL for a default name.

The new view is placed relative to the existing view specified by the *referenceViewId* parameter. If this parameter is set to the constant RootWindowPlace, then placement is made relative to the root window of the Windowing System (i.e., relative to the full display screen in most cases). If this parameter is set to the constant MainViewPlace, the new view is placed relative to the main Altia interface view. This would be the first view opened by the interface. All other values are taken to be the id number of a view previously opened using AtOpenSimpleView/altiaOpenSimpleView, AtOpenView/altiaOpenView, or AtOpenWindowView/altiaOpenWindowView.

The *referenceX* and *referenceY* parameters specify the pixel position of the new view's lower left corner relative to the reference view's lower left corner. For instance, passing a *referenceX* of 20 and *referenceY* of 20 would position the new view's lower left corner 20 pixels to the right and 20 pixels above the reference

view's lower left corner. The new view can be centered within the reference view by passing the constants **CenterViewX** and **CenterViewY** for *referenceX* and *referenceY*.

NOTE: Window Manager borders around the new view may affect the position of the new view slightly.

The caller may also specify the area of the design that should appear within the view. The parameters *universeX* and *universeY* specify the design universe coordinates that are to appear in the lower left corner of the new view. Please see AtMoveView, altiaMoveView for more details on determining design universe coordinates. The width and height of the view window as well as the magnification factor that should be applied to the design universe may also be specified through the *width*, *height*, and *magnification* parameters, respectively. Lower left corner coordinates, size, or magnification of the view may be left up to the Altia interface system by using the constants **DefaultViewX**, **DefaultViewY**, **DefaultViewHeight**, **DefaultViewWidth**, or **DefaultViewMag** for *universeX*, *universeY*, *height, width* and *magnification*, respectively.

AtOpenViewPlaced is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaOpenViewPlaced acts on the currently active Altia interface connection.

3.11.4 Return Values

If a communications failure occurs, these functions return -1. If an illegal parameter is detected, -2 is returned. Otherwise, 0 is returned. Not all illegal parameters can be detected prior to sending the open request to the Altia interface. The Altia interface does not return an error to the program if it receives an illegal open request. It simply will not open the new view.

3.11.5 See Also

AtOpenSimpleView, altiaOpenSimpleView, AtOpenView, altiaOpenView, AtOpenWindowView, altiaOpenWindowView, AtCloseView, altiaCloseView

3.12 AtOpenWindowView, altiaOpenWindowView

3.12.1 Name

AtOpenWindowView	Toolkit function to open a new view into an existing window.
altiaOpenWindowView	Base library version of AtOpenWindowView.

3.12.2 C Synopsis

```
int AtOpenWindowView(AtConnectId connectId, int windowId, const char *viewName,
                     int universeX, int universeY, double magnification);

int altiaOpenWindowView(int windowId, const char *viewName,int universeX,
                        int universeY, double magnification);
```

3.12.3 Description

Both functions send a request to the Altia interface to open a new view into an existing Windowing System window previously created by the calling application or some other application which is permitting the calling application to access the window. The position of objects within the view and magnification factor of the view can be specified.

The window id of the existing window is passed via the *windowId* parameter. This id is also used as a parameter to other view manipulation routines. The window id must be for a window that is associated with the same display that the Altia interface is associated with. For displays running the X Windowing System, *windowId* is a window id returned from a XCreateWindow or equivalent call. If drawing or refreshing of other graphics is necessary within the window, it must be handled by the application; however, this degree of window sharing is not recommended because it is difficult to control the order in which the processes sharing the window will receive exposure events.

The *viewName* parameter provides a method for specifying a name for the new view. The name will appear in the Window Manager banner portion of the view. The *viewName* is typically entered as a double-quoted string such as "Extra View" or NULL for a default name.

The caller may specify the area of the design that should appear within the view. The parameters *universeX* and *universeY* specify the design universe coordinates that are to appear in the lower left corner of the new view. Please see AtMoveView, altiaMoveView for more details on determining design universe coordinates. The magnification factor that should be applied to the design universe may also be specified through the *magnification* parameter. Lower left corner coordinates or magnification of the view may be left up to the Altia interface system by using the constants **DefaultViewX**, **DefaultViewY**, or **DefaultViewMag** for *universeX*, *universeY*, and *magnification*, respectively.

AtOpenWindowView is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaOpenWindowView acts on the currently active Altia interface connection.

3.12.4 Return Values

If a communications failure occurs, these functions return -1. If an illegal parameter is detected, -2 is returned. Otherwise, 0 is returned. Not all illegal parameters can be detected prior to sending the open request to the Altia interface. The Altia interface does not return an error to the program if it receives an illegal open request. It simply will not open the new view.

3.12.5 See Also

AtCloseView, altiaCloseView

3.13 AtSetDesignId, altiaSetDesignId

3.13.1 Name

AtSetDesignId	Toolkit function to set the currently active design file id.
altiaSetDesignId	Base library version of AtSetDesignId.

3.13.2 C Synopsis

```
int AtSetDesignId(AtConnectId connectId, int designId);

int altiaSetDesignId(int designId);
```

3.13.3 Description

Both functions set the current design id for an Altia interface to the value given by *designId*. The current design id must be correctly set when using routines that require an object id (such as the clone and object move routines) because object id's are only unique within a design file. The current design id must also be correctly set for the various view open and manipulation routines so that views are opened and changed for the correct design file.

The initial value of the design id is always 0 which implies all cloning, object moving, and view opens/changes will be directed to the original design file if AtSetDesignId and altiaSetDesignId are never called to change the design id.

AtSetDesignId is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaSetDesignId acts on the currently active Altia interface connection.

3.13.4 Return Values

If a communications failure occurs, these functions return -1. If an illegal parameter is detected, -2 is returned. Otherwise, 0 is returned.

3.13.5 See Also

AtOpenDesignFile, altiaOpenDesignFile, AtCloseDesignId, altiaCloseDesignId

3.14 AtSizeView, altiaSizeView

3.14.1 Name

AtSizeView	Toolkit function to change the size of a view.
altiaSizeView	Base library version of AtSizeView.

3.14.2 C Synopsis

```
int AtSizeView(AtConnectId connectId, int viewId, int width, int height);

int altiaSizeView(int viewId, int width, int height);
```

3.14.3 Description

Both functions send a request to the Altia interface to change the size of a view previously opened with:

- **AtOpenSimpleView/altiaOpenSimpleView**
- **AtOpenView/altiaOpenView**
- **AtOpenViewPlaced/altiaOpenViewPlaced**
- **AtOpenWindowView/altiaOpenWindowView.**

The identification number of the view to change is passed in *viewId*. You can change the size of the default Main View by passing a *viewId* of 0. The *width* and *height* parameters specify the view's new width and height in pixels.

AtSizeView is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaSizeView acts on the currently active Altia interface connection.

3.14.4 Return Values

If a communications failure occurs, these functions return -1. If *width* or *height* is less than or equal to 0, -2 is returned. Otherwise, 0 is returned. An illegal view id cannot be detected prior to sending the size request to the Altia interface. In such cases, the return value will be 0 and the size operation will simply not happen.

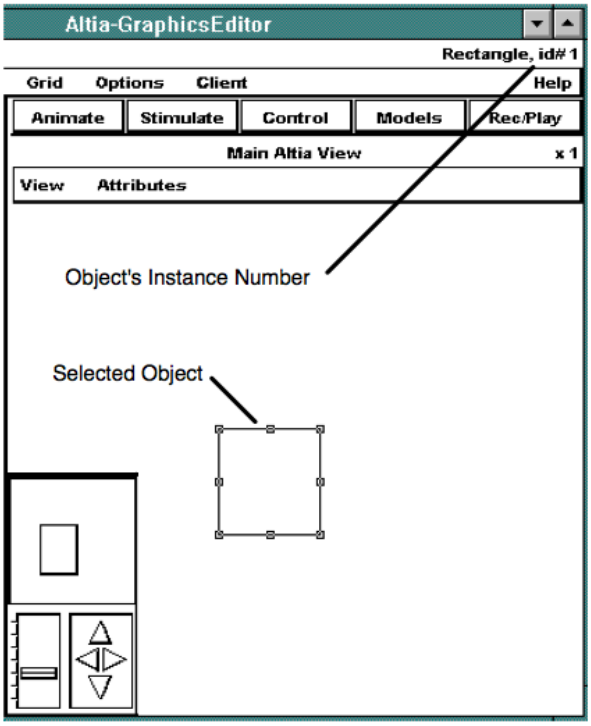
4 Routines for Creating and Manipulating Clones

4.1 Introduction

This section describes Altia library and toolkit routines that allow for the program- matic creation of one or more copies of an existing Altia interface object. This is re- ferred to as cloning and the new copies are called clones. A clone is completely independent of the original object.

When a clone is created by the Altia interface system, its animation and stimulus function names are automatically prepended with "#:" to uniquely identify them from the original object's function names. A clone is created by calling AtCreateClone or altiaCreateClone. A clone number is selected by the program and passed as a pa- rameter to either function. The '#' in "#:" is the ASCII representation of the selected clone number (e.g., if AtCreateClone is passed a clone number of 9 and the object be- ing cloned has an animation named "move", the clone will have an animation named "9:move" that will behave in an identical fashion, but with respect to the clone in- stead of the original object). All of the routines supplied by this library that deal with clone function names take or receive the clone number separated from the base an- imation or stimulus function name. As such, it is actually unnecessary for an appli- cation developer to know that the clone's animation and stimulus functions are derived using the "#:name" syntax. But, it is certainly helpful to know this when you are studying a design within the Altia Graphics Editor that has clones in it.

In addition to requiring a clone number, AtCreateClone and altiaCreateClone must also know the identity of the object that is to be cloned. This identity is established by passing the original object's *instance* number as a parameter. Each object in an Altia interface design has a unique number associated with it. This number is created automatically when the object is originally drawn or copied into a drawing area of the Altia Graphics Editor. To determine an object's number, select the object in the Editor and read the object status information in the Editor's upper right corner. See the figure below for an illustration.



If a design is displayed in the Altia Graphics Editor and an application program is connected to the interface and creating clones, the clones are considered temporary objects by the Editor. If the design is saved to a file (using the **Save** or **Save As** option from the **File** menu), the clones are *not* saved. In the process of debugging design and program interaction, it is often desirable to delete all existing clones. This is done by simply selecting the **Delete All Clones** option from the **Client** menu of the Editor.

To provide complete backwards compatibility, the clone creation and manipulation routines are provided in base library and toolkit form. Toolkit versions begin with the *At* prefix and base library versions use an *altia* prefix. Programs using cloning routines must link with the *UNIX* liblan.a object library found in the lib or libfloat directory of the Altia software installation. On systems running *Microsoft Windows*, programs are linked with libdde.lib, or liblan.lib if TCP/IP socket software and hardware exists. On all types of systems, the include file for the library is named altia.h and it resides in the same directory as the object libraries.

The Altia Design software installation provides a demos directory containing various complete Altia interface examples. The files found in the clones sub-directory provide a working example of clone creation and manipulation from an application program. See the README file in the **clones** directory for more details. The map sub-directory contains a more sophisticated example and has a README file as well which you may consult for more details. As a quick summary, here is a table of the clone manipulation routines:

Routine	Description
AtCreateClone	Creates a copy of an existing Altia interface object.

Routine	Description
AtSendCloneEvent	Sends an animation event to a clone object.
AtSendCloneText	Sends text to a cloned Altia text output or text input object.
AtGetCloneText	Gets text from a cloned Altia text input object.
AtAddCloneCallback	Adds a callback procedure to receive clone events of a particular type.
AtRemoveCloneCallback	Removes a clone callback procedure from the callback list.
AtCloneProc	Prototype for a clone event callback procedure.
AtMoveClone	Repositions a clone object.
AtDeleteClone	Deletes a clone object.
AtDeleteAllClones	Deletes all clones from the Altia interface.
AtCheckCloneEvent	Checks for a specific clone event from the Altia interface.
AtCloneEventSelected	Checks if a specific clone event is selected for receipt by the program.
AtPollCloneEvent	Get the current value for a specific clone animation.
altiaCreateClone	Base Library version of AtCreateClone.
altiaSendCloneEvent	Base Library version of AtSendCloneEvent.
altiaSendCloneText	Base Library version of AtSendCloneText.
altiaGetCloneText	Base Library version of AtGetCloneText.
altiaMoveClone	Base Library version of AtMoveClone.
altiaDeleteClone	Base Library version of AtDeleteClone.
altiaDeleteAllClones	Base Library version of AtDeleteAllClones.
altiaCheckCloneEvent	Base Library version of AtCheckCloneEvent.
altiaCloneEventSelected	Base Library version of AtCloneEventSelected.
altiaPollCloneEvent	Base Library version of AtPollCloneEvent

4.2 AtAddCloneCallback, AtRemoveCloneCallback, AtCloneProc

4.2.1 Name

AtAddCloneCallback	Toolkit function to register a procedure as a callback for a clone event.
AtRemoveCloneCallback	Toolkit function to remove a clone event callback procedure.
AtCloneProc	Prototype for a clone event callback procedure

4.2.2 C Synopsis

```
void AtAddCloneCallback(AtConnectId connectId, int cloneNumber,
                        const char *eventName, AtCloneProc callbackProc,
                        AtPointer clientData);

void AtRemoveCloneCallback(AtConnectId connectId, int cloneNumber,
                           const char *eventName, AtCloneProc callbackProc,
                           AtPointer clientData);

typedef void (*AtCloneProc)(AtConnectId connectId, int cloneNumber,
                            const char *eventName, AltiaEventType eventValue,
                            AtPointer clientData);
```

4.2.3 Description

AtAddCloneCallback is a toolkit routine that adds a procedure to the Altia interface event callback list for an event that is being generated by a clone object. When an Altia interface event is received from the connection specified by *connectId* and the event's clone number matches *cloneNumber* and the event's name matches the string pointed to by *eventName*, the callback procedure given by *callbackProc* will be invoked. AtAddCloneCallback makes the appropriate call to AtSelectEvent to insure that the client application receives events of the type specified by *eventName* from the clone object specified by *cloneNumber*. Special data may be passed to the callback procedure through the *clientData* parameter, or it can be set to NULL. Callbacks can only be invoked if the application program executes AtMainLoop; or, if it has its own main loop, it must call AtDispatchEvent when an Altia interface event is received with AtNextEvent.

AtRemoveCloneCallback is a toolkit routine that removes a clone callback procedure from the Altia interface event callback list if it is currently in the list. A procedure is only removed if it was added to the list with exactly the same attributes specified by *connectId*, *cloneNumber*, *eventName*, *callbackProc*, and *clientData*. If there are no other callbacks in the list for the given clone event, AtUnselectEvent will be called to turn off receipt of the events from the Altia interface to improve performance.

AtCloneProc specifies the arguments and return value that a clone callback procedure must have in order for it to be compatible with the toolkit. The clone callback procedure will receive the id of the connection from which the event originated (*connectId*), the number of the clone originating the event (*cloneNumber*), the name of the event (*eventName*), the value for the event (*eventValue*), and the custom data that was passed to AtAddCloneCallback when the callback was registered (*clientData*).

The type for *eventValue*, AltiaEventType, is an int if the integer libraries are used and a double if the floating point libraries are used.

4.2.4 Return Values

None

4.2.5 Example

The following C main routine opens an Altia interface connection, creates a clone, adds a clone callback for event **press**, then executes the toolkit main loop to wait for **press** events. When a **press** event is received by the clone callback function, it tests the value for the event to determine what action it should take. In this example, the custom data field for the callback is unused.

```
#include "altia.h"

void pressHandler();

#define ORIGINALOBJECT 1762  /* Object # for object being cloned */

main(argc, argv)
int argc;
char *argv[];
{
    AtConnectId connectId;
    int cloneNumber = -1;  /* Start with clone number -1 */
    if ((connectId = AtOpenConnection(NULL, NULL, argc, argv)) < 0)
    {
        fprintf(stderr, "AtOpenConnection failed\n");
        exit(1);
    }
    AtCreateClone(connectId, ORIGINALOBJECT, cloneNumber);
    AtAddCloneCallback(connectId, cloneNumber, "press", pressHandler, NULL);
    AtMainLoop();
}

void pressHandler(connectId, cloneNumber, eventName, eventValue, data)
AtConnectId connectId;
int cloneNumber;
char *eventName;
AltiaEventType eventValue;
AtPointer data;
{
    if (eventValue == 1)
        printf("Received a press event from clone %d\n", cloneNumber);
    else
        printf("Received an unpress event from clone %d\n", cloneNumber);
}
```

4.2.6 See Also

AtMainLoop, AtCreateClone, altiaCreateClone

4.3 AtCheckCloneEvent, altiaCheckCloneEvent

4.3.1 Name

AtCheckCloneEvent	Toolkit function to check for a pending clone event with a specific name.
altiaCheckCloneEvent	Base library version of AtCheckCloneEvent.

C Synopsis

```
int AtCheckCloneEvent(AtConnectId connectId, int cloneNumber,
                      const char *eventName, AltiaEventType *eventValueOut);

int altiaCheckCloneEvent(int cloneNumber, const char *eventName,
                         AltiaEventType *eventValueOut);
```

4.3.2 Description

Either function is called to get the value for the next available clone event that was delivered to the program by the Altia interface and whose name matches *eventName*. If there are no clone events available with the given name for the clone specified by *cloneNumber*, both functions return immediately. If one or more clone events are queued with the specified name, the value for the oldest event is returned and that instance of the event is removed from the pending event queue. All other events, including newer instances of the named clone event, remain in the queue in their appropriate order. In order for these functions to work correctly, the clone event name must be selected for receipt prior to calling AtCheckCloneEvent or altiaCheckCloneEvent. If AtAddCloneCallback was called to register a callback for the clone event name, then it is automatically selected for receipt.

The *eventValueOut* parameter must point to an AltiaEventType variable. If an event is found, its value is returned in the AltiaEventType pointed to by *eventValueOut*. The type for *eventValueOut*, AltiaEventType, is an int if the integer libraries are used and a double if the floating point libraries are used.

Normally, events are received by calling AtNextEvent or altiaNextEvent. However, these functions get the next available event for any name and they block waiting for a new event if one is not immediately available. AtCheckEvent or altiaCheckEvent can be used in conjunction with AtNextEvent or altiaNextEvent, or by themselves to accommodate different program architectures.

AtCheckCloneEvent is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaCheckCloneEvent acts on the currently active Altia interface connection.

4.3.3 Return Values

If an event is found, 1 is returned and the integer pointed to by *eventValueOut* is set to the event's value. If no event is found, 0 is returned. If a communications failure occurs, -1 is returned.

4.3.4 See Also

AtNextEvent, altiaNextEvent

4.4 AtCloneEventSelected, altiaCloneEventSelected

4.4.1 Name

AtCloneEventSelected	Toolkit function to check if a specific clone event has been selected for receipt.
altiaCloneEventSelected	Base library version of AtCloneEventSelected.

4.4.2 C Synopsis

```
int AtCloneEventSelected(AtConnectId connectId, int cloneNumber,
                         const char *eventName);

int altiaCloneEventSelected(int cloneNumber, const char *eventName);
```

4.4.3 Description

Either function is called to check if clone events of the given name have been selected for receipt by the program. A clone event name is selected for receipt automatically when AtAddCloneCallback is called to register a clone callback for the name. The *eventName* parameter points to the name of the event that should be checked while *cloneNumber* gives the number of the clone.

These functions are useful for large, modular application programs. If one module wishes to determine if a particular clone event name has been selected for receipt by another module, it can call AtCloneEventSelected or altiaCloneEventSelected to do so.

AtCloneEventSelected is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaCloneEventSelected acts on the currently active Altia interface connection.

4.4.4 Return Values

If the clone event name is selected for receipt, 1 is returned. If the clone event name is not selected for receipt, 0 is returned. If a communications failure occurs, -1 is returned.

4.4.5 See Also

AtAddCloneCallback, AtRemoveCloneCallback, AtCloneProc, AtCheckCloneEvent, altiaCheckCloneEvent

4.5 AtCreateClone, altiaCreateClone

4.5.1 Name

AtCreateClone	Toolkit function to create a duplicate of an existing Altia interface object.
altiaCreateClone	Base library version of AtCreateClone.

4.5.2 C Synopsis

```
int AtCreateClone(AtConnectId connectId, int objectNumber, int cloneNumber);

int altiaCreateClone(int objectNumber, int cloneNumber);
```

4.5.3 Description

Both functions send a request to the Altia interface to create a clone of an existing interface object. The existing object is identified by its instance number which is passed as the *objectNumber* parameter (see *Introduction* on page 4-1, for details on determining an object's instance number). The number for the clone is determined by the caller and passed as the *cloneNumber* parameter. The new clone's number must be unique with respect to all existing clone numbers which have already been created. For example, if a clone already exists with a clone number of 1, then a second clone can be created with a clone number of 2 or any number other than 1. A clone number may be reused after a clone has been deleted with AtDeleteClone/altiaDeleteClone, AtDeleteAllClones/altiaDeleteAllClones, or from the Altia Graphics Editor using **Cut** from the Tool Panel or the **Delete All Clones** option from the **Client** menu.

After creation, the cloned object is positioned identically to the original object and on top of it. To move a clone object to a new location, use the AtMoveClone or altiaMoveClone function after the clone is created. If the original object is in a group, the clone is created within the same group. The original object can be a primitive object (e.g., rectangle, square, circle, etc.) or a group object.

If the original object has animation or stimulus definitions, those definitions are automatically renamed with the format "#:original_name" where # is the ASCII equivalent of *cloneNumber* and original_name is the original name of the animation or stimulus. This allows programs to send or receive events to/from the specific cloned object without affecting the original object. All of the routines supplied by this library that deal with clones take or receive the clone number separated from the original animation or stimulus name. As such, it is actually unnecessary for an application developer to know that the clone's animation and stimulus names are derived using the "#:original_name" syntax. But, it is certainly helpful to know this when you are studying a design within the Altia Graphics Editor that has clones in it.

AtCreateClone is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaCreateClone acts on the currently active Altia interface connection.

4.5.4 Return Values

If a communications failure occurs, these functions return -1. Otherwise, 0 is returned. The Altia interface does not return an error to the program if it receives an illegal *cloneNumber* or *objectNumber* value. It simply will not create a clone.

4.5.5 Warnings

The value of *cloneNumber* may be any integer value of your choosing. As a good practice, however, it is recommended that negative values be used for clone numbers. For instance, start with -1 and decrement the value as new clones are created. Use of negative values for clone numbers is not a requirement, but it insures that clone object numbers are unique from normal object numbers. Conflicts can only occur if programs are creating clone objects and also manipulating normal objects with the AtMoveObject/altiaMoveObject routines. This is a defect which should be fixed in a future release.

If an application program is creating, deleting, and recreating clones in Altia Design 2.1 or earlier, the numbers for clones which have been deleted should be reused for new clones rather than simply incrementing or decrementing a count ad infinitum. The second approach will cause slow, but persistent memory growth to occur within the Altia Graphics Editor or Altia Runtime system. This defect was fixed in Altia Design 2.2.

4.5.6 See Also

AtDeleteClone, AtDeleteAllClones, altiaDeleteClone, altiaDeleteAllClones, AtMoveClone, altiaMoveClone

4.6 AtDeleteClone, AtDeleteAllClones, altiaDeleteClone, altiaDeleteAllClones

4.6.1 Name

AtDeleteClone	Toolkit function to delete a specific clone object from an Altia interface.
altiaDeleteClone	Base library version of AtDeleteClone.
AtDeleteAllClones	Toolkit function to delete all clone objects from an Altia interface.
altiaDeleteAllClones	Base library version of AtDeleteAllClones.

4.6.2 C Synopsis

```
int AtDeleteClone(AtConnectId connectId, int cloneNumber);

int altiaDeleteClone(int cloneNumber);

int AtDeleteAllClones(AtConnectId connectId);

int altiaDeleteAllClones();
```

4.6.3 Description

AtDeleteClone and altiaDeleteClone send a request to the Altia interface to delete the clone object with the clone number specified by *cloneNumber*. After a clone is deleted, its clone number may be reused to create a new clone of an existing Altia interface object.

AtDeleteAllClones and altiaDeleteAllClones send a request to the Altia interface to delete all of the clone objects in the interface. All clone numbers may be reused after the call.

AtDeleteClone and AtDeleteAllClones are the toolkit versions of the functions. Each takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library versions altiaDeleteClone and altiaDeleteAllClones act on the currently active Altia interface connection.

4.6.4 Return Values

If a communications failure occurs, these functions return -1. Otherwise, 0 is returned. The Altia interface does not return an error to the program if it receives an illegal *cloneNumber* value. It simply will not delete the specified clone (however, the typical problem is that no such clone exists so there is nothing to delete).

4.6.5 Warnings

If an application program is creating, deleting, and recreating clones in Altia Design 2.1 or earlier, the numbers for clones which have been deleted should be reused for new clones rather than simply incrementing or decrementing a count ad infinitum. The second approach will cause slow, but persistent memory growth to occur within the Altia Graphics Editor or Altia Runtime system. This defect was fixed in Altia Design 2.2.

4.6.6 See Also

AtCreateClone, altiaCreateClone

4.7 AtGetCloneText, altiaGetCloneText

4.7.1 Name

AtGetCloneText	Toolkit function to get a text string from a clone of an Altia text i/o object.
altiaGetCloneText	Base library version of AtGetCloneText.

4.7.2 C Synopsis

```
int AtGetCloneText(AtConnectId connectId, int cloneNumber, const char *textName,
                   char *buffer, int bufferSize);

int altiaGetCloneText(int cloneNumber, const char *textName, char *buffer,
                      int bufferSize);
```

4.7.3 Description

Both functions get the text string currently being displayed by a text i/o object clone.

First, some background on Altia text i/o objects is in order. Altia text i/o objects allow Altia interface designs to accept keyboard input from users. The input is displayed in a text field and dynamically updated as the user types in characters from the computer's keyboard. Text i/o objects have a built-in animation, referred to as the "text" animation, which can be queried at any time by an application program. A query causes the text i/o object to send its stored text as a string of "text" events. The events can be received by the program and stored in a character buffer. Several styles of text i/o objects are available from the Inputs, Textio, and GUI Models Libraries. These are design files, inputs.dsn, textio.dsn. and gui.dsn or guiwin.dsn to be precise, found in the **models** sub-directory of the Altia software installation. They are organized for display within an Altia Models View (press the **Models** button in the Altia Design Editor to initiate the opening of a Models View). To learn specific details about text i/o objects, select the **Textio Models** option from the **Help** menu found in any Altia Models View.

AtGetCloneText and altiaGetCloneText query the text i/o object clone specified by *cloneNumber* for its displayed text. The name of the "text" animation to use in the query is pointed to by *textName*. The name could simply be "text" or it could be something else if the original object's "text" animation was renamed using the Altia Animation Rename Dialog. The text that is retrieved is stored in the character buffer pointed to by *buffer* as a null terminated string (i.e., ends in '\0'). Note that the caller must supply a buffer - it is *not* supplied by the routines. The character size of the buffer is specified by *bufferSize*. If the string returned by the object is larger than *bufferSize*, then it is truncated and null terminated to fit in the character buffer.

The clone number given by *cloneNumber* is actually prepended to the animation name pointed to by *textName* to create a clone animation name of the form "#:text_name". "#" is the string representation of *cloneNumber* and "text_name" is the name in the string pointed to by *textName*. AtGetCloneText and altiaGetCloneText are actually just wrappers around AtGetText and altiaGetText to relieve the programmer from the tedious task of concatenating clone numbers to animation names. See *Introduction* for more details on the clone animation name format.

AtGetCloneText is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaGetCloneText acts on the currently active Altia interface connection.

Designs created prior to software release 2.0 could only use the obsoleted (but still supported) text input object instead of the newer and more versatile text i/o object. The text input object has a "getchar" animation instead of a "text" animation. It is queried using AtGetCloneText in the same was as the "text" animation.

4.7.4 Return Values

If a communications failure occurs or the "text" animation for the clone object does not respond to a query, these functions return -1. In either case, the first element of the array pointed to by *buffer* is set to the null character ('\0'). A 0 is returned if the query was successful and the array pointed to by *buffer* will contain a null terminated string. If the input object is not displaying any text, the array will only contain a null character.

4.7.5 Example

The following C code shows how AtGetCloneText can be used with one of the text input areas from the Inputs, Textio, or GUI Models Libraries. For software release 2.0 or later, text input areas from these libraries generate a "done" event when a user finishes text input. Text input is finished by pressing the Return/Enter key or by clicking the left mouse button outside of the input area. This example registers a callback for the "done" clone event. When the callback is invoked, it queries the "text" animation for the text i/o object clone to get the user's text input.

```
#include "altia.h"

void doneHandler();

#define ORIGINALOBJECT 1762 /* Object # of object being cloned */

main(argc, argv)
int argc;
char *argv[];
{
    AtConnectId connectId;
    int cloneNumber = -1;  /* Start with clone number -1 */
    if ((connectId = AtOpenConnection(NULL, NULL, argc, argv)) < 0)
    {
        fprintf(stderr, "AtOpenConnection failed\n");
        exit(1);
    }
    AtCreateClone(connectId, ORIGINALOBJECT, cloneNumber);
    AtAddCloneCallback(connectId, cloneNumber, "done", doneHandler, NULL);
    AtMainLoop();
}

void doneHandler(connectId, cloneNumber, eventName, eventValue,data)
AtConnectId connectId;
int cloneNumber;
char *eventName;
AltiaEventType eventValue;
AtPointer data;
{
```

```
        char buffer[50];
        /* This code assumes there is a second text i/o object
        * in the interface with a "text" animation named "out_text" and
        * it sends the input text to it or an error message.
        */
        if (AtGetCloneText(connectId, cloneNumber, "text",buffer,50) == 0)
            AtSendText(connectId, "out_text", buffer);
        else
            AtSendText(connectId,"out_text","Error - AtGetCloneText failed");
    }
```

4.7.6 See Also

AtSendCloneText, altiaSendCloneText, AtGetText, altiaGetText

4.8 AtMoveClone, altiaMoveClone

4.8.1 Name

AtMoveClone	Toolkit function to reposition a clone object within an Altia interface.
altiaMoveClone	Base library version of AtMoveClone.

4.8.2 C Synopsis

```
int AtMoveClone(AtConnectId connectId, int cloneNumber, int xOffset, int yOffset);

int altiaMoveClone(int cloneNumber, int xOffset, int yOffset)'
```

4.8.3 Description

Both functions move the Altia interface clone object identified by *cloneNumber* from its current position within the interface to a new position. The new position is *xOffset* pixels from the original position in the x direction and *yOffset* pixels from the original position in the y direction (i.e., the move is relative to the object's current position). *xOffset* or *yOffset* is positive to achieve a move to the right or up, respectively. *xOffset* or *yOffset* is negative to achieve a move to the left or down, respectively. All animation and stimulus for the object will be performed relative to the object's new position. Note that move animation can be used to move an object rather than using these Altia library functions. The functions, however, allow for more freedom of movement.

AtMoveClone is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaMoveClone acts on the currently active Altia interface connection.

4.8.4 Return Values

If a communications failure occurs, these functions return -1. Otherwise, 0 is returned. The Altia interface does not return an error to the program if it receives an illegal *cloneNumber* value. It simply will not move the specified clone (however, the typical problem is that no such clone exists so there is nothing to move).

4.9 AtPollCloneEvent, altiaPollCloneEvent

4.9.1 Name

AtPollCloneEvent	Toolkit function to get the current state of an Altia interface clone animation.
altiaPollCloneEvent	Base library version of AtPollCloneEvent.

4.9.2 C Synopsis

```
int AtPollCloneEvent(AtConnectId connectId, int cloneNumber, const char *eventName,
                     AltiaEventType *eventValueOut);

int altiaPollCloneEvent(int cloneNumber, const char *eventName,
                        AltiaEventType *eventValueOut);
```

4.9.3 Description

Both functions get the current state of an Altia interface clone event for the clone identified by *cloneNumber*. The event's name is given by *eventName* and its current state is returned in the AltiaEventType variable pointed to by *eventValueOut*. The type for *eventValueOut*, AltiaEventType, is an int if the integer libraries are used and a double if the floating point libraries are used.

If any clone events with the same name were previously routed to this client application and they are awaiting receipt via AtNextEvent, altiaNextEvent or AtMainLoop, the events will be deleted from the receiving queue. A poll request gets the current value for the event - this makes all previous values obsolete.

These functions are special versions of the AtPollEvent and altiaPollEvent functions described in Section 2, General Library Routines. Their behavior is similar to the functions described in Section 2, General Library Routines, except that they take an additional parameter - cloneNumber - for referencing a clone event name instead of a normal event name.

AtPollCloneEvent is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaPollCloneEvent acts on the currently active Altia interface connection.

4.9.4 Return Values

If there is no Altia interface clone animation event with the given name or a communications failure occurs, these functions return -1. Otherwise, 0 is returned.

4.9.5 See Also

AtSendCloneEvent, altiaSendCloneEvent

4.10 AtSendCloneEvent, altiaSendCloneEvent

4.10.1 Name

AtSendCloneEvent	Toolkit function to send an animation event to an Altia interface clone object.
altiaSendCloneEvent	Base library version of AtSendCloneEvent.

4.10.2 C Synopsis

```
int AtSendCloneEvent(AtConnectId connectId, int cloneNumber, const char *eventName,
                     AltiaEventType eventValue);

int altiaSendCloneEvent(int cloneNumber, const char *eventName,
                        AltiaEventType eventValue);
```

4.10.3 Description

Both functions initiate the transmission of an animation event to an Altia interface. The clone number given by *cloneNumber* is prepended to the event name pointed to by *eventName* to create a clone animation name of the form "#:event_name". "#" is the string representation of *cloneNumber* and "event_name" is the name in the string pointed to by *eventName*. The value for the event is specified by *eventValue*. The type for *eventValue*, AltiaEventType, is an int if the integer libraries are used and a double if the floating point libraries are used.

These functions are just wrappers around AtSendEvent and altiaSendEvent to relieve the programmer from the tedious task of concatenating clone numbers to event names when dealing with clone animation events. See *Introduction* for more details on the clone animation name format.

AtSendCloneEvent is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaSendCloneEvent acts on the currently active Altia interface connection.

A call to AtSendCloneEvent or altiaSendCloneEvent forces a state change of an Altia interface animation function. This usually results in a visual change to a clone object within the Altia interface.

4.10.4 Return Values

If a communications failure occurs, these functions return -1. Otherwise, 0 is returned.

4.10.5 Windows Dependencies

To improve inter-process communications performance, events from AtSendCloneEvent and altiaSendCloneEvent calls are buffered on the application's side until a synchronous call such as AtNextEvent/altiaNextEvent or AtPollEvent/altiaPollEvent is performed or the program returns to AtMainLoop from a callback if AtMainLoop is being used. If this behavior produces undesirable effects, a call to AtFlushOutput/altiaFlushOutput can be made to force buffered events to be sent immediately. Or, turn off buffering permanently with a call to AtCacheOutput/altiaCacheOutput.

4.10.6 See Also

AtSendCloneText, altiaSendCloneText

4.11 AtSendCloneText, altiaSendCloneText

4.11.1 Name

AtSendCloneText	Toolkit function to send a string of animation events to an Altia clone object.
altiaSendCloneText	Base library version of AtSendCloneText.

4.11.2 C Synopsis

```
int AtSendCloneText(AtConnectId connectId, int cloneNumber, const char *eventName,
                    const char *text);

int altiaSendCloneText(int cloneNumber, const char *eventName, const char *text);
```

4.11.3 Description

Both functions initiate the transmission of a string of animation events to an Altia interface. The values for the events are the values of the characters provided by the string pointed to by *text*. The *text* string must be terminated with the null character ('\0'). The values, including the terminating null character, are sent in sequence to the animation function given by *eventName* for the clone specified by *cloneNumber*.

The clone number given by *cloneNumber* is actually prepended to the event name pointed to by *eventName* to create a clone animation name of the form "#:event_name". "#" is the string representation of *cloneNumber* and "event_name" is the name in the string pointed to by *eventName*.

AtSendCloneText and altiaSendCloneText are actually just wrappers around AtSendText and altiaSendText to relieve the programmer from the tedious task of concatenating clone numbers to event names when dealing with clone animation events. See *Introduction* for more details on the clone animation name format.

AtSendCloneText and altiaSendCloneText are normally used for transmitting character strings to Altia interface text i/o object clones. Altia text i/o objects have built-in animation, referred to as the "text" and "character" animations, that accept events and treat them as text characters. The characters are displayed on the screen in the font style and color chosen by the interface designer. Several styles of text i/o objects are available from the Inputs, Textio, and GUI Models Libraries. These are design files, inputs.dsn, textio.dsn and gui.dsn or guiwin.dsn to be precise, found in the models sub-directory of the Altia software installation. They are organized for display within an Altia Models View (press the **Libraries** button in the Altia Design Editor to initiate the opening of a Models View). To learn specific details about text i/o objects, select the **Textio Models** option from the **Help** menu found in any Altia Models View.

AtSendCloneText is the toolkit version of the function. It takes a connection id as returned by AtOpenConnection or AtStartInterface. The base library version altiaSendCloneText acts on the currently active Altia interface connection.

Designs created prior to software release 2.0 could only use the obsoleted (but still supported) text input and output objects instead of the newer and more versatile text i/o object. The obsoleted text input and output objects have a "putchar" animation instead of a "text" and "character" animation.

4.11.4 Return Values

If a communications failure occurs, these functions return -1. Otherwise, 0 is returned.

4.11.5 Windows Dependencies

To improve inter-process communications performance, events from AtSendCloneText and altiaSendCloneText calls are buffered on the application's side until a synchronous call such as AtNextEvent/altiaNextEvent or AtPollEvent/altiaPollEvent is performed or the program returns to AtMainLoop from a callback if AtMainLoop is being used. If this behavior produces undesirable affects, a call to AtFlushOutput/altiaFlushOutput can be made to force buffered events to be sent immediately. Or, turn off buffering permanently with a call to AtCacheOutput/altiaCacheOutput.

4.11.6 See Also

AtSendCloneEvent, altiaSendCloneEvent, AtGetCloneText, altiaGetCloneText

5 Library Routines for Widget Integration

This section describes Altia toolkit routines that allow for the integration of an Xt (X toolkit) widget interface and an Altia interface. The approach used is to have the Xt application create a drawing area widget in its own hierarchy of widgets and then request that the Altia interface use the drawing area widget as if it were one of its own view windows.

The widget integration functions are only provided in toolkit form on UNIX Systems. Programs using these toolkit routines need to link with the liblan.a object library. In addition to including the standard altia.h header file, programs also need to include altiaXt.h.

In addition to these manual pages, the **widget** sub-directory of the Altia Design installation **demos** directory provides a working example of an integration of an Altia interface and an Xt widget application. As a quick summary, here is a table of the routines provided:

Routine	Description
AtCreateMotifWidget	Creates a *Motif* drawing area widget and sets it up to be used as an Altia view.
AtCreateOpenlookWidget	Creates an *Openlook* stub widget and sets it up to be used as an Altia view.
AtXtOpenConnection	Opens an Altia interface connection like AtOpenConnection does, but also performs some *Xt* specific tasks to insure that the connection works in concert with *Xt*.
AtXtAppOpenConnection	Opens an Altia interface connection like AtOpenConnection does, but also performs some Xt specific tasks to insure that the connection works in concert with Xt.
AtUnrealizeWidget	Unrealizes a widget being used as the window for an Altia view.

5.1 AtCreateMotifWidget

5.1.1 Name

AtCreateMotifWidget	Create a Motif drawing area widget for use as an Altia view.

5.1.2 C Synopsis

```
Widget AtCreateMotifWidget(Widget parent, const char *name, ArgList args,
                           Cardinal count)
```

5.1.3 Description

This Altia toolkit function returns a created *Motif* DrawingArea widget configured for use as an Altia view. The parent of the new widget is specified by *parent* and *name* points to a string containing the widget's chosen name. Arguments for the widget are passed via the *args* parameter and the number of arguments is given by *count*. In addition to all of the arguments available for a DrawingArea widget, the following additional arguments are recognized:

XmNmagnification	Allows for the setting of the view's initial magnification. The value passed is the desired magnification value of the view multiplied by 100 (e.g., for a 2x magnification, the value would be 200). If **XmNmagnification** is not supplied, then the Altia interface chooses a magnification based on resource settings from the Runtime Configuration File, *X* resource database, *X* defaults files, Altia application defaults file, or built-in defaults.
XmNxloc **XmNyloc**	Allows for specification of the x,y coordinates of the Altia drawing universe that should occupy the lower left corner of the view. If either **XmNxloc** or **XmNyloc** are supplied, both should be supplied. If **XmNxloc** and **XmNyloc** are not supplied, then the Altia interface chooses a x,y pair based on resource settings from the Runtime Configuration File, *X* resource database, *X* defaults files, Altia application defaults file, or built-in defaults. For details on determining universe coordinates, consult the Altia view manipulation routine AtMoveView.
XmNclientConnection	Allows for the specification of an Altia interface connection id. The value passed is the connection id (type AtConnectId) for the Altia interface that will be drawing into the new widget. If this argument is not set, then the default connection at the time this widget is realized will be used.

To change the magnification factor or universe x,y coordinates of a view after it is realized, simply use the appropriate Altia view manipulation routine (AtMagnifyView or AtMoveView) and pass the *X* Window id as the view id.

NOTE: The DrawingArea's *userData* variable is used by the Altia library. Do not attempt to set it - things will not work correctly if you do.

The widget created by AtCreateMotifWidget is an unmanaged widget and therefore requires a call to XtManageChild.

5.1.4 Return Values

A widget id is returned if this call was successful. For other return values, consult documentation for XmCreateDrawingArea. AtCreateMotifWidget is simply a wrapper around that function.

5.1.5 Example

See the complete on-line demonstration in the **widget** sub-directory of the Altia Design installation **demos** directory.

5.2 AtCreateOpenlookWidget

5.2.1 Name

AtCreateOpenLookWidget	Create an *OpenLook* stub widget for use as an Altia view.

C Synopsis

```
Widget AtCreateOpenLookWidget(Widget parent, const char *name, ArgList args,
                              Cardinal count)
```

5.2.2 Description

This Altia toolkit function returns a created *OpenLook* Stub widget configured for use as an Altia view. The parent of the new widget is specified by *parent* and *name* points to a string containing the widget's chosen name. Arguments for the widget are passed via the *args* parameter and the number of arguments is given by *count*. In addition to all of the arguments available for a Stub widget, the following additional arguments are recognized:

XmNmagnification	Allows for the setting of the View's initial magnification. The value passed is the desired magnification value of the view multiplied by 100 (e.g., for a 2x magnification, the value would be 200). If **XmNmagnification** is not supplied, then the Altia interface chooses a magnification based on resource settings from the Runtime Configuration File, X resource database, X defaults files, Altia application defaults file, or built-in defaults.
XtNxloc **XtNyloc**	Allows for specification of the x,y coordinates of the Altia drawing universe that should occupy the lower left corner of the view. If either **XtNxloc** or **XtNyloc** are supplied, both should be supplied. If **XtNxloc** and **XtNyloc** are not supplied, then the Altia interface chooses a x,y pair based on resource settings from the Runtime Configuration File, X resource database, X defaults files, Altia application defaults file, or built-in defaults. For details on determining universe coordinates, consult the Altia view manipulation routine AtMoveView.
XtNclientConnection	Allows for the specification of an Altia interface connection id. The value passed is the connection id (type AtConnectId) for the Altia interface that will be drawing into the new widget. If this argument is not set, then the default connection at the time this widget is realized will be used.

To change the magnification factor or universe x,y coordinates of a view after it is realized, simply use the appropriate Altia view manipulation routine (AtMagnifyView or AtMoveView) and pass the X Window id as the view id.

NOTE: The Stub's *userData* variable is used by the Altia library. Do not attempt to set it - things will not work correctly if you do.

The widget created by AtCreateOpenlookWidget is an unmanaged widget and therefore requires a call to XtManageChild.

5.2.3 Return Values

A widget id is returned if this call was successful. For other return values, consult documentation for XtCreateWidget. AtCreateOpenlookWidget is simply a wrapper around a call to XtCreateWidget to create a widget of class stubWidgetClass.

5.2.4 Example

See the complete on-line demonstration in the **widget** sub-directory of the Altia Design installation **demos** directory.

5.3 AtUnrealizeWidget

5.3.1 Name

AtUnrealizeWidget	Altia function to unrealize a widget that is being used as an Altia view.

5.3.2 C Synopsis

```
void AtUnrealizeWidget(Widget widget)
```

5.3.3 Description

This Altia function should be called instead of XtUnrealizeWidget to unrealize a widget created with AtCreateMotifWidget or AtCreateOpenlookWidget. Unrealizing a widget destroys the widget's window. The Altia interface using the widget as a window for an Altia view must know that the window id is no longer valid. If any parent of this widget is unrealized, then the AtUnrealizeWidget routine should be called as well because this will also unrealize the widget.

If the widget is realized again at some later time, then the new window id will be automatically registered with the Altia interface as an Altia View window.

AtUnrealizeWidget does not need to be called on program termination because Altia will detect the closing of the Altia connection socket and it will automatically disassociate itself with the program's windows.

5.3.4 Return Values

None

5.3.5 See Also

XtUnrealizeWidget

5.4 AtXtOpenConnection, AtXtAppOpenConnection

5.4.1 Name

AtXtOpenConnection	Open a connection to an Altia interface for use within an X*t* application.
AtXtAppOpenConnection	Like AtXtOpenConnection, but for Xt applications using app contexts.

5.4.2 C Synopsis

```
AtConnectId AtXtOpenConnection(const char *portName, const char *optionName,
                               int argc, const char *argv[]);

AtConnectId AtXtAppOpenConnection(XtAppContext context, const char *portName,
                                  const char *optionName, int argc,
                                  const char *argv[]);
```

5.4.3 Description

These Altia toolkit functions are Xt (X toolkit) versions of AtOpenConnection. See AtOpenConnection, AtCloseConnection for specific descriptions for the *portName*, *optionName*, *argc*, and *argv* parameters.

AtXtOpenConnection makes a call to AtOpenConnection followed by a call to XtAddInput to include the Altia connection's file descriptor as an additional input source for Xt. This allows the use of Altia toolkit callback routines in conjunction with Xt's main loop functions and callbacks.

AtXtAppOpenConnection is for use in Xt applications that are using application contexts to control multiple Xt interfaces. An application context is passed via *context* and it is in turn used in a call to XtAppAddInput.

Please not that Altia toolkit timer callbacks are inoperative when the Altia toolkit is used in this manner. Instead, use the Xt timer callback facilities - they are functionally equivalent.

If Altia callbacks are not used in conjuction with AtXtOpenConnection or AtXtAppOpenConnection, then X and Altia events must be manually processed using XtPending with XtProcessEvents rather than using XtMainLoop for automated processing. If XtPending returns XtIMAlternateInput, then call AtNextEvent to get Altia events. Otherwise, call XtProcessEvents to process X events.

5.4.4 Return Values

Identical to the return values for AtOpenConnection.

5.4.5 Example

See a complete on-line demonstration in the **widget** sub-directory of the Altia Design installation **demos** directory.

5.4.6 See Also

AtOpenConnection, AtCloseConnection