

Unicode User's Guide

Altia® Design 10.2 for Windows®

Altia Deepscreen®

HMI Software

July, 2012



Please read these important notices before using this release.

Beginning with Altia Design 10.2, the default editor is a Unicode editor. Designs containing high/extended ASCII text characters and/or Unicode text characters must not be opened and saved in a non-Unicode editor.

A Unicode editor saves high/extended ASCII text characters (character values 128 to 255) and Unicode text characters (character values 256 to 65535) to a design (.dsn) file in a multi-byte format. This format is not recognized by any non-Unicode editors in earlier releases of Altia Design. A non-Unicode editor processes text characters only as single bytes. A non-Unicode editor will incorrectly display these special characters when it opens a design file that contains them. For example, it displays 2 or 3 unexpected byte characters for each special character.

If the design is saved from a non-Unicode editor, these special characters are permanently replaced with the 2 or 3 unexpected single byte characters. The original special characters are only recovered by manually changing a Label, Text I/O or MLTO to show the special characters instead of the 2 or 3 unexpected single byte characters.

Recent Altia Design releases save design files in a new format.

A design (.dsn) file saved in Altia Design 9.0 or newer has a new format and will not open in earlier versions of Altia Design, Altia Runtime, or Altia FacePlate. Please back-up existing design files before saving to the same files with Altia Design 9.0 or newer. Doing so allows you to open a back-up version of a design in an earlier release if desired.

The FreeType font engine incorporated into Altia Design 10.0 and newer releases may change how individual characters, words, and spaces are rendered.

Projects created in Altia Design 9.2 or earlier releases that rely on very precise text placement should not be migrated to this release of Altia Design incorporating the FreeType font engine.

1.0 About this document	1
2.0 Preparing Altia Design 10.2 for Unicode Development	1
2.1 Associate the Unicode Design Editor with the .dsn File Extension.....	2
2.2 Choosing a Font in Altia Design for Unicode Characters	3
3.0 Designing with a Unicode Version of Altia Design	5
3.1 Converting a Design Created in a Non-Unicode Version of Altia Design ...	5
3.2 Use the Font Chooser to Set a Unicode Font for Objects that will Display Unicode Characters.....	5
3.3 Entering International Characters from another Application	5
3.4 Characters for the High/Extended Range of the 8-bit ASCII Character Code Set	6
3.5 Entering International Characters Directly into a Design	7
4.0 DeepScreen Code Generation with a Unicode Version of Altia Design	9
4.1 Generating and Compiling Code from a Unicode Version of Altia Design..	9
4.2 Generating and Compiling altiaGL or Accelerated Target Code for Unicode	10
5.0 Using the Altia API to Interface Code to a Unicode Design	13
5.1 Documentation to Study before Reading this Section	13
5.2 Integrating Code with Unicode Compatible DeepScreen Code	13
5.3 API Header File Supports Portable Application Code	13
5.4 Using <code>AtStartInterface()</code> in a Unicode Version of the API.....	16
5.5 Compiling Code to Use the Unicode Version of the Altia API	16
5.6 Wide String Manipulation Functions on Targets without Support	16

1.0 About this document

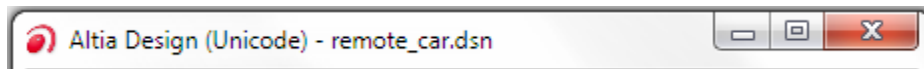
This document describes how to develop for Unicode with Altia Design 10.2 to run in Altia Design, Altia Runtime, and Altia DeepScreen generated code. This document assumes that the Altia software is installed in `c:\usr\altia`. If your installation directory is different, occurrences of `c:\usr\altia` would be replaced with your chosen installation directory.

In addition to this document, the Altia Design *Enhancements Summary* document and *DeepScreen User's Guide* describe general enhancements and new features not specifically related to Unicode support for Altia Design and DeepScreen, respectively. The *Altia Design 10 User's Guide* and *Altia API 10.0 Reference Manual* are also good sources for more product information. To view these documents from Altia Design, open the **Help** menu and choose the **Enhancements Summary...**, **DeepScreen User's Guide...**, **User's Guide...** or **Altia API Reference Manual...** option.

2.0 Preparing Altia Design 10.2 for Unicode Development

As of Altia Design 10.2, the Unicode editor is the default editor.

From an Altia Design 10.2 or newer program group under the Windows **Start > All Programs > Altia > Altia Design** menu, select the **Altia Editor** item to start the Altia Design editor. For Altia Design 10.2 or newer, the **Altia Editor** item now refers to the Unicode version of the Altia Design editor. The Unicode version of the Altia Design editor is now the only available editor. It can open, display, and save all existing design (`.dsn`) files that were saved with a non-Unicode version of the Altia Design editor. You can recognize when a Unicode Altia Design editor is running because the banner at the top of the editor's window identifies it as a Unicode editor. The picture below shows the top banner for a Unicode Altia Design editor running with the design file `remote_car.dsn` opened.



To enable code generation of Unicode (wide character) DeepScreen code, there is now a **Generate Unicode Font Characters** option in the Code Generation Options dialog. This option is explained in more detail in the *DeepScreen User's Guide* (open the *DeepScreen User's Guide* from the Altia Design **Help** menu). If this option is not enabled, the DeepScreen generated code is for 8-bit character support like the generated code from the previously available non-Unicode versions of the Altia Design editor.

Do Not Open/Save Designs Containing High/Extended ASCII Text Characters or Unicode Text Characters in a Non-Unicode Altia Design Editor

A Unicode editor saves high/extended ASCII text characters (character values 128 to 255) and Unicode text characters (character values 256 to 65535) to a design (`.dsn`) file in a multi-byte format. This format is not recognized by any non-Unicode editors from earlier releases of Altia Design. A non-Unicode editor processes text characters only as single bytes. A non-Unicode editor will incorrectly display these special characters when it opens a design file that contains them. For example, it displays 2 or 3 unexpected byte characters for each special character. If the design is saved from the non-Unicode editor, the special characters are permanently replaced with the 2 or 3 unexpected single byte characters. The original special characters are only recovered by manually changing Label objects, Text I/O objects or Multi-line Text objects (MLTOs) to show the special characters instead of the 2 or 3 unexpected single byte characters.

The executable programs for Altia Design and Runtime (altia.exe for the Unicode version of Altia Design and altiart.exe for the Unicode version of Altia Runtime) are installed in the Altia Design software bin directory. This is c:\usr\altia\bin if the software is installed in c:\usr\altia.

If you have not already done so, choose the **Altia Editor** icon from the **Altia Design 10.2** program group.

Perhaps you want to copy this **Altia Editor** icon to the Windows desktop for easy access. To do so, right click on the icon when it is highlighted and select **Copy** from the menu. Then right click in empty space on the Windows desktop and select **Paste shortcut** from the menu. A double-click on the new **Altia Editor** desktop icon will always start the Unicode version of the Altia Design 10.2 editor.

To open a design file in a Unicode Altia Runtime, pass the design file name to altiart.exe from the Altia Design 10.2 or newer software bin directory. For example, from a Windows BAT script or Command Prompt window, do:

```
c:\usr\altia\bin\altiart.exe my_design_file.dsn
```

If there is application code that uses the Altia API function AtStartInterface() to programmatically start Altia Runtime, it knows how to execute altiart.exe. For Altia Design 10.2 or newer, altiart.exe from the Altia Design software bin folder is the Unicode version of Altia Runtime. Create a copy of altiart.exe in the working directory of the application (that is, the folder from which the application executable is started). Also copy the fonts.ali and colors.ali files from the Altia software bin directory to the application's working directory. When the application executable starts and calls the AtStartInterface() Altia API function, the AtStartInterface() function always looks in the working directory for a copy of altiart.exe and executes it with the specified design file. If this copy is the Unicode Altia Runtime, it executes it just the same and this provides the ability to display extended Unicode characters in an Altia Runtime controlled by an application program. There is more work to do in the application program to send/receive Unicode characters to/from Altia. This is described in a later section of this document [Using the Altia API to Interface Application Code to a Unicode Design](#).

2.1 Associate the Unicode Design Editor with the .dsn File Extension

If you make frequent use of double-clicking on a .dsn design file (while it is showing in a Windows Explorer or My Computer window) to open Altia Design, it will open the Unicode version of Altia Design if Altia Design 10.2 or newer is the most recently installed Altia Design software.

If a design containing Unicode characters is saved in an 8-bit character version of Altia Design, the Unicode characters are permanently lost!

If the Unicode version of Altia Design is not associated with the .dsn file extension, it is a good idea to change the association. To change a file type association on Windows 7, open the **Start** menu, type **change file type** in the search box and then select the **Change the file type associated with a file extension** result. This opens the **Control Panel->Programs->Default Programs->Set Associations** dialog. Here you can highlight the .dsn extension from a list of all known file type associations and then press the **Change Program...** button to change its association. Press the **Browse...** button to browse to the Altia Design 10.2 or newer software bin folder and choose the altia.exe executable. Work back through the **Open**, **OK** and **Close** buttons to apply the change.

To change a file type association on Windows XP, open a Windows Explorer or My Computer window, open its **Tools** menu and choose the **Folder Options...** option. Select

the **File Types** tab in the Folder Options dialog. Arrange the listing by File Types (click on the **File Types** label under **Registered file types**), scroll until the **Altia Design File** type is shown, highlight it and press the **Advanced** button. Double-click on the **open** action in the new Edit File Type dialog. Fix the path to `altia.exe` in the **Application used to perform action** field so that it refers to the `altia.exe` executable in an Altia Design 10.2 or newer software `bin` folder. Press the **OK** button when the changes are complete. Press **OK** in the Edit File Type dialog and press **OK** in the Folder Options dialog.

At a later date, if a different version of the Altia Design software is installed, the association for the `.dsn` extension will change to the newly installed software's Altia Design editor. If this is not a Unicode Altia Design editor, repeat the above steps for Windows 7 or Windows XP to change the association back to a Unicode Altia Design if desired.

2.2 Choosing a Font in Altia Design for Unicode Characters


The fonts available from the Altia Design 8.0 or newer Font Chooser drop-down list do not support extended Unicode characters in general (such as Chinese, Japanese, Korean). A custom font must be chosen. Versions of Microsoft Office 2003 or newer include an excellent Unicode font **Arial Unicode MS**. The **Arial Unicode MS** font supports Unicode characters for a wide variety of languages. This is unlike most other Unicode fonts that typically only support the 0-255 ASCII/ANSI character codes and the Unicode character codes for one other region/language (for example, Japanese, but not Chinese or Korean). If you do not have a license of Microsoft Office 2003 or newer installed on the computer, the **Arial Unicode MS** font can be purchased separately. As of the writing of this document, Ascender Fonts (<http://www.ascendercorp.com/font/arial-unicode/>) sells licenses for **Arial Unicode MS**.


NOTE: The proper licensing of any font, such as for use in Altia DeepScreen generated code for an embedded device, is the responsibility of each Altia Design and Altia DeepScreen user and not the responsibility of Altia, Inc. Please also note that for generating DeepScreen code to an embedded device, the **Arial Unicode MS** font may not be a good choice because it supports so many characters. The resulting code will probably be too large (10s of Mbytes or more) for the memory of the device. An option is to limit the generation of character glyphs using a custom DeepScreen `.gen` file. How to do this is beyond the scope of this document. Please open an existing `.gen` file for the DeepScreen target of interest for possible examples.



If you think your computer has **Arial Unicode MS** installed, but you are not sure, open the **Fonts** folder from the **Control Panel** (on Windows 7, it is **Control Panel->Appearance and Personalization->Fonts**). The fonts are listed in alphabetical order so **Arial Unicode MS** should be near the top of the list.

If you have a Unicode font as a file (such as a file with a `.ttf` extension) and want to install it on your computer, just copy and paste the file into the **Fonts** folder of the **Control Panel** (on Windows 7, it is **Control Panel->Appearance and Personalization->Fonts**). This automatically installs it on the computer for all applications to use. On Windows 7, be sure that its **Show/hide** status is set to **Show** in the **Fonts** folder.

If a Unicode font such as **Arial Unicode MS** is installed, here are the steps to assign this font to an Altia object that displays text (such as a static Unicode Label object, Text I/O object, or Multi-line Text object):

1. With Unicode Altia Design in **Edit** mode, select a text item in the work area or a Group containing a text item. A text item is a static Unicode Label object created from the **Text Tool** , a Text I/O object copied from a model library, or

a Multi-line Text object (MLTO) copied from a model library. Please note that static Label objects created in a non-Unicode 8-bit character version of Altia Design cannot handle extended Unicode characters (just Western 8-bit characters) even when the design containing the static Label is opened in a Unicode Altia Design session. Only a Unicode Label object created from the **Text Tool**  in a Unicode Altia Design session can handle extended Unicode characters.

2. With the object still selected from step #1, activate the **Text Tool**  if it is not already the active tool. The Font Chooser  appears in the toolbar area above the work area and it displays the font name/size/attributes for the currently selected object. If the object is a Group that has never had a font chosen for it, the Font Chooser just display a default font name/size/attributes. Type the name of the Unicode font into the Font Chooser name field (such as **Arial Unicode MS**), change the point size if desired, and change any of the other available attributes if desired (such as making text Bold or Italic). If a Group is selected, the new font name/size/attributes are associated with the Group which forces the font name/size/attributes on all child objects of the Group.
3. If you are not sure of the font name to use in step #2, confirm the font name from the **Fonts** folder in the **Control Panel** (on Windows 7, it is **Control Panel->Appearance and Personalization->Fonts**). Just double-click on the font's description as it appears in the **Fonts** folder and a property dialog opens. The name in the **Typeface name:** field (or **Font name:** field on Windows 7) is the name to use in the Altia Font Chooser.

3.0 Designing with a Unicode Version of Altia Design

READ THIS WARNING BEFORE USING THIS SOFTWARE

NEVER open a design containing Unicode characters into a non-Unicode version of Altia (Altia Design) and then save the design! The Unicode characters will be permanently lost! They cannot be recovered without manually changing them back to Unicode characters in a Unicode version of Altia Design.


Also please note that a Unicode version of Altia Design does not support DeepScreen code generation for literal Unicode strings in any Altia control statements (for example, `SET 1_text = "Unicode chars here"`). If strings containing Unicode characters must be written to dynamic Text I/O objects at execution time of DeepScreen generated code, it must be done from C application code using the Altia API functions `AtSendText()` / `altiaSendText()`.

3.1 Converting a Design Created in a Non-Unicode Version of Altia Design

Any Altia .dsn design file created with a non-Unicode version of Altia Design will open into a Unicode version.

Existing Text I/O objects and Multi-Line Text objects (MLTOs) can automatically take Unicode characters.

Existing static text (Label) objects, however, will NOT accept Unicode characters.

Any static text object that must take Unicode characters needs to be replaced using the **Text Tool**  on the Tool-Command Panel of a Unicode version of Altia Design.

You can differentiate a Unicode static text object from a regular static text object by the information that appears at the bottom of the editor when the object is selected in a Unicode version of Altia.

If a static text object is Unicode capable, the object information is **[Unicode Label, id# XX]**. If a static text object is NOT Unicode capable (because it was created in a non-Unicode version of Altia Design or the design was accidentally saved in a non-Unicode version of Altia Design), the object information is simply **[Label, id# XX]**.

IMPORTANT WARNING!

After saving a design in a Unicode version, NEVER open and save the design in any non-Unicode version. All Unicode capable Label objects will be converted back into 8-bit Label objects! 8-bit Label objects cannot display Unicode characters.

3.2 Use the Font Chooser to Set a Unicode Font for Objects that will Display Unicode Characters

See the earlier section [Choosing a Font in Altia Design for Unicode Characters](#) for detailed steps to set the font on an object in the work area.

3.3 Entering International Characters from another Application

Using another Windows application such as the Character Map utility program is an easy way to select specific Unicode characters and copy/paste them into a design opened in Altia Design. On Windows XP or Windows 7, open the Character Map program by selecting the **Start** menu, go to **All Programs, Accessories, System Tools** and finally choose **Character Map**. The **Help** button in Character Map explains how to use the tool.

After one or more characters are copied to the clipboard from another application such as Character Map, return to the Altia Design editor and use `Ctrl+V` to paste the characters. To paste them into a static text Label or Text I/O object, begin editing the object (double-click on it in the Altia work area) before pressing `Ctrl+V`. If the Text I/O object has properties, you can also paste into the property that is showing the current text.

IMPORTANT NOTE

The input of international characters should be limited to text that appears in the design itself. Absolutely do not create animation names, stimulus names, control names or control variables that use international characters, spaces or tabs, or even ASCII symbols besides underscore (`_`). Enter international characters for a property if the property is for setting the text of a dynamic Text I/O object or Multi-Line Text object (MLTO).

Also please note that a Unicode version of Altia Design does not support DeepScreen code generation for literal Unicode strings in any Altia control statements (for example, `SET l_text = "Unicode chars here"`). If strings containing Unicode characters must be written to dynamic Text I/O objects at execution time of DeepScreen generated code, it must be done from C application code using the Altia API functions `AtSendText()`/`altiaSendText()`.

3.4 Characters for the High/Extended Range of the 8-bit ASCII Character Code Set

IMPORTANT NOTE

Unfortunately, the capability described here is not working in Altia Design 7.0 or newer. Please use the Character Map utility for all special character entry as described in the previous topic [Entering International Characters from another Application such as the Character Map Utility](#).

To enter characters with character code values in the range 128-255, press the keyboard ALT key and hold it down while pressing 0 (zero) from the numeric keypad followed by the 3 decimal digits of the character code value from the numeric keypad. **Only enter the numbers (0 and then the 3 decimal digits) from the keyboard's numeric keypad (on the far right side of most keyboards).** For most configurations, the Num Lock mode must be off.

For example, the Yen symbol is character code value 165. To enter it into an Altia text field, use:

ALT+0165

On a laptop with no explicit numeric keypad, there is usually a *virtual* numeric keypad that shares physical keys with other characters on the keyboard. How to enable the *virtual* numeric keypad may vary with each laptop model. Try holding the Function (Fn) key and the ALT key simultaneously while entering 0 (zero) and the 3 decimal digits of the character code value from the *virtual* numeric keypad. As with a regular keyboard, the Num Lock mode may need to be off for this to work.

The font currently chosen in the font chooser must support the high/extended range of the 8-bit ASCII character code set. Most standard Windows fonts do this and the code values follow the ISO8859-1 character set standard. If the currently selected font does

not support a certain character in the extended range, the character will most likely appear as a rectangle (either filled or unfilled).

3.5 Entering International Characters Directly into a Design

This topic describes the process of entering international characters directly into a design based on Windows 2000. Other types of Windows platforms (for example, Windows XP and Windows 7) probably require different techniques for entering international characters directly into an application.

IMPORTANT NOTE

The input of international characters should be limited to text that appears in the design itself. Absolutely do not create animation names, stimulus names, control names or control variables that use international characters. Enter international characters for a property if the property is for setting the text of a dynamic Text I/O object or Multi-Line Text object (MLTO).

Also please note that this Unicode version of Altia Design does not support DeepScreen code generation for literal Unicode strings in any Altia control statements (for example, `SET l_text = "Unicode chars here"`). If strings containing Unicode characters must be written to dynamic Text I/O objects at execution time of DeepScreen generated code, it must be done from C application code using the Altia API functions `AtSendText()`/`altiaSendText()`.

1. You would typically start by installing one or more Windows 2000 MultiLanguage extensions (for Japanese, Korean, Chinese or other language regions of interest).
2. After a MultiLanguage extension is installed (say for Japanese) on Windows 2000, open the Control Panel and select Regional Options.

In the **General** tab of the Regional Options dialog, you can set **Your locale (location)**: to any available location. However, this is typically not necessary. Unlike the earlier Far East versions of Altia Design, the Unicode versions do not depend on this local setting.

The more important setting is in the **Input Locales** tab. Highlight the desired **Input language** and press the **Set as Default** button.

Press the **OK** button to apply these changes and exit the Regional Options dialog.

1. Start the Unicode version of Altia Design. As soon as Altia opens and is the active window, a new floating task bar should appear in the lower-right corner of the display.

This new floating task bar will typically only appear when one of the Altia editor windows is the active window.

How the task bar works depends on the input language chosen in the previous step. Here are some examples:

Japanese

The left most icon of the floating bar shows the current input mode. If you leave the mouse cursor over it for a few seconds, a bubble should appear saying Input Mode: Direct Input.

For entering regular text in any Altia input field (such as the Animation Editor, Stimulus Editor or Control Editor), the input mode should be Direct Input. An

uppercase A in the icon indicates this mode. If the mouse cursor lingers over the icon, the bubble appears with the phrase Input Mode: Direct Input.

To enter Japanese characters for a text object in the Altia drawing area, the input mode must be Hiragana. To toggle the input mode, press Left+Alt+~ (i.e., hold the Alt key on the left side of the keyboard and then press the ~ key). In Hiragana input mode, the input mode icon changes from A to a Japanese character. . If the mouse cursor lingers over the icon, the bubble appears with the phrase Input Mode: Hiragana.

In Hiragana input mode, you can type regular phrases on the keyboard and it attempts to convert these into Japanese characters. A temporary window appears in the upper-left corner of the display to provide conversion options for the characters being entered. We will assume a user familiar with Japanese language support on Windows knows how to control this conversion process.

Korean

The left most icon of the floating bar shows the current input mode. If the letter A is showing in the icon, the input mode is English. To switch the input mode for entering Korean characters, right click on the icon and choose Hangul mode. Return to English by right clicking and choosing the English mode option.

In Hangul mode, type regular characters on the keyboard and a conversion window appears showing the possible conversion to a Korean character. The conversion window behaves differently than the conversion window for Japanese so experiment with pressing the Space bar or Enter key to see how it responds.

Chinese

The left most icon of the floating bar shows the current input mode. If the letter A is showing in the icon, the input mode is English. To switch the input mode for entering Chinese characters, position the mouse cursor directly over the icon and press the Shift key on the keyboard.

In Chinese character input mode, type regular characters on the keyboard and a conversion window appears showing the possible conversion to a Chinese character. The conversion window behaves differently than the conversion windows for the other languages so experiment with pressing the Space bar or Enter key to see how it responds.

4.0 DeepScreen Code Generation with a Unicode Version of Altia Design

IMPORTANT WARNING!

After saving a design in a Unicode version, NEVER open and save the design in any non-Unicode (standard or legacy Far East) version. All Unicode capable Label objects will be converted back into 8-bit Label objects! 8-bit Label objects cannot display Unicode characters.

Also please note that a Unicode version of Altia Design does not support generating code for literal Unicode strings in any Altia control statements (for example, `SET l_text = "Unicode chars here"`). If strings containing Unicode characters must be written to dynamic Text I/O objects at execution time of DeepScreen generated code, it must be done from C application code using the Altia API functions `AtSendText()` / `altiaSendText()`.

4.1 Generating and Compiling Code from a Unicode Version of Altia Design

To generate DeepScreen code for a design that displays Unicode characters, it is necessary to use a Unicode version of Altia Design.

For the most part, the generated code is identical to code generated from a standard version of Altia Design.

If, however, a Unicode Label object, dynamic Text I/O object, or Multi-Line Text object (MLTO) contains one or more Unicode characters, the string constant generated to `altia/data.c` for the object resembles:

```
#ifndef UNICODE
ALT_TEXT("näçnäûnäÄnäÄnäinäæöñΘçî")
#else
ALT_TEXT("\x3107\x3116\x310f\x310e\x310d\x3111\x5931\x96cc")
#endif
```

If the UNICODE compile flag is NOT set at compile time, the string will contain a multi-byte version of the character codes which will most likely look very strange as it does in the above sample.

If the UNICODE compile flag is set at compile time, the string contains a sequence of Unicode character values between 0 and 65535 with each value represented using the C escape sequence `\XXXX`. This allows the declaration of a literal character value (in hex) within a string constant. In addition, `ALT_TEXT` is a C preprocessor macro for the C language L cast. The C compiler actually gets the following line as its input:

```
L"\x3107\x3116\x310f\x310e\x310d\x3111\x5931\x96cc"
```

The compiler should convert this to a string of wide (Unicode) characters, each character containing the next hex value, and a final wide character with a value of 0 to identify the end of the string. MSVC as well as GCC work this way.

If a Label, Text I/O, or MLTO contains NO Unicode characters (i.e., all the characters are ASCII values in the range 0 to 127), the string constant written to `altia/data.c` for the object is simply something like:

```
ALT_TEXT("Hello World")
```

If the UNICODE compile flag is set at compile time, ALT_TEXT is a C preprocessor macro for the C language L cast. The C compiler actually gets the following line as its input:

```
L"Hello World"
```

The compiler should convert this to a string of wide (Unicode) characters, each character containing an ASCII value between 0 and 127 to represent the next ASCII character in the string, and a final wide character with a value of 0 to identify the end of the string. MSVC as well as GCC work this way.

To learn how to set the UNICODE compile flag at compile time, continue to the next topic.

4.2 Generating and Compiling altiaGL or Accelerated Target Code for Unicode

The following are specific details for generating and compiling altiaGL or Accelerated target code for Unicode, and are only applicable for a software installation licensed to generate code for the DeepScreen altiaGL target or an accelerated target (e.g., Fujitsu Jade). If you are new to DeepScreen code generation, it may help to read the *DeepScreen User's Guide* before reading this topic. To view the DeepScreen User's Guide from Altia Design, open the **Help** menu and choose the **DeepScreen User's Guide...** option.

The important component for Unicode code generation and compiling is the **Generate Unicode Font Characters** option in the Code Generation Options dialog. This option is not supported by every DeepScreen target. Please see a target's separate User's Guide for details. For DeepScreen targets that support the **Generate Unicode Font Characters** option, generating and compiling code for Unicode is relatively simple. Start the Unicode version of Altia Design, open a design file, open the **Code Generation...** menu, and choose the **Generate Source Code...** option. In the Code Generation Options window, choose the desired target for which to generate code and choose the desired code gen options along with the **Generate Unicode Font Characters** option. Choose a **Makefile script:** file by pressing the associated **Browse...** button and picking one of the listed template Makefile scripts. Choose an appropriate **Additional Target files list:** file using its associated **Browse...** button. Finally, press the **Generate** button. The selected template Makefile script is generated to `altmake.bat` in the same directory containing the opened design file. Compile the generated code according to the documentation specific to the target.

If a target does not have an existing Unicode template Makefile script, the following can help you to create a Unicode template Makefile script.

1. Create a template Makefile script for generating Unicode compatible code for the target.

Begin by making a copy of a working template Makefile script for the same target. This is typically done in the DeepScreen target's software installation folder. The location of this folder varies from target to target. Please see the *DeepScreen User's Guide* for help. For example, the folder location could be `C:\usr\AltiaDeepScreen\sw_renderwin32_542\ds_altiagl_win32\src` for the 5.4.2 version of the Intel x86 Software Render Windows target. Give the new copy a name that suggests its purpose. For example, if the target is a model xyz device, a good name for the file might be:

```
altmake_xyz_unicode.bat
```

Determine what data type the target system requires for Unicode characters. How to do this for a given target is beyond the scope of this document. If necessary, contact the provider of the target development tools for help. With this information available, study the following conditional code (which was extracted from the DeepScreen target software installation `ds_engine/src/altiaBase.h` file) to determine which compiler macro needs to be defined so that the Altia generated code uses the correct data type for Unicode characters on the target:

```
#ifndef UNICODE
typedef char ALTIA_CHAR;
#elif defined(WIN32) || defined(WCHAR_IS_USHORT)
typedef unsigned short ALTIA_CHAR;
#elif defined(WCHAR_IS_UINT)
typedef unsigned int ALTIA_CHAR;
#elif defined(WCHAR_IS_ULONG)
typedef unsigned long ALTIA_CHAR;
#elif defined(WCHAR_IS_SHORT)
typedef short ALTIA_CHAR;
#elif defined(WCHAR_IS_LONG)
typedef long ALTIA_CHAR;
#else
typedef int ALTIA_CHAR;
#endif
```

For example, if the target Unicode character data type is unsigned long, `WCHAR_IS_ULONG` must be defined at compile time. No extra define is needed if the Unicode data type is just int.

Edit (with a simple text editor) the new template Makefile script, locate the definition for `TARGETDEFS`, and add:

```
-DUNICODE -D_UNICODE -DWCHAR_IS_XXXX
```

Replace `XXXX` with an appropriate type (for example `ULONG` if the target Unicode data type is unsigned long). No `-DWCHAR_IS_XXXX` is needed if the Unicode data type is just int.

Also modify the assignment for `APIUNICODEDEF` to be:

```
APIUNICODEDEF=-DALTIAUNICODEAPI
```

Save the modifications to the new template Makefile script.

2. Remember the location of the new template Makefile script. It can be kept in the same DeepScreen target software installation folder as the target's other template Makefile scripts (this makes it easy to locate with the **Browse...** button for the **Makefile script:** field in the Code Generation Options dialog). It can also be kept in any other folder accessible to the computer (this just adds more steps to browsing for the file).
3. Start a Unicode version of Altia Design and open a design file. From Altia's **Code Generation** menu, choose the **Generate Source Code...** option. Select the desired target and code gen options from the Code Generation Options window and also set the file name in the **Makefile script:** field to the name of the new template Makefile script created in steps #1 and #2.
4. For example, if the new template Makefile script is named `altmake_xyz_unicode.bat` for the Intel x86 Software Render Windows target

installed in C:\usr\AltiaDeepScreen\sw_renderwin32_542, you would set the **Makefile script:** field to:

```
C:\usr\AltiaDeepScreen\sw_renderwin32_542\ds_altiagl_win32\src  
\altmake_xyz_unicode.bat
```

Press the **Generate** button to generate the code.

5. Compile the generated code as usual for the target.

The C compiler must accept the C language `L` cast before a string constant and properly convert the string's characters to the Unicode character data size supported by the target.

For the Altia generated code, it is the `ALT_TEXT()` C preprocessor macro that applies the `L` cast. The `ALT_TEXT()` macro's definition is located in the Altia `usercode/altiaBase.h` file. For Unicode compiling, the definition is:

```
#define ALT_TEXT(x) L ## x
```

The `##` is understood by the C preprocessor as a request to concatenate `L` with `x`. The C preprocessor takes a source line like:

```
ALT_TEXT("Hello World")
```

and translates it into the following source line for the C compiler:

```
L"Hello World"
```

The macro definition can be customized for compilers that do not understand this syntax (although it seems to be widely supported).

5.0 Using the Altia API to Interface Code to a Unicode Design

This section is focused on how to use the Altia API to develop application code for interfacing with Unicode DeepScreen generated code. These concepts also generally apply to developing application code to interface with a design running in a Unicode version of Altia Design or Altia Runtime. The difference is that application code interfacing to a Unicode version of Altia Design or Altia Runtime links with a Unicode version of the Altia API object library. For Windows, this is a .lib file from the Altia software lib/ms32 or libfloat/ms32 folder that has Unicode in its name. Examples are libddeUnicode.lib or liblanUnicode.lib. Except for this difference, the coding techniques described in this section are identical for application code interfacing to Unicode DeepScreen generated code, Unicode Altia Design, or Unicode Altia Runtime.

5.1 Documentation to Study before Reading this Section

This section is not intended to describe the complete details of using the Altia API to communicate with application code. Users that are new to the idea of integrating application code with Altia graphics can study the following Altia documents to help them understand the concepts of the Altia API and how to use it:

[Altia API Reference Manual, Chapter 0 – Getting Started with Code Links](#)

[Altia API Reference Manual, Chapter 2 – General Library Routines](#)

[Altia Design C Code Tutorial](#)

[DeepScreen User's Guide, Chapter 3, Section 8 – DeepScreen Version of C/C++ API Limitations](#)

[DeepScreen User's Guide, Chapter 7 – Integrating DeepScreen Code into Another Application](#)

To view any one of these documents from Altia Design Unicode, open the **Help** menu in the Graphics Editor and choose the appropriate option.

5.2 Integrating Code with Unicode Compatible DeepScreen Code

In typical application code that uses the Altia API to communicate with either Altia Design/Runtime or DeepScreen generated code, character strings passed into API functions or returned by API functions refer to 8-bit ASCII characters.

If the Altia API is compiled for Unicode, character strings passed into API functions or returned by API functions refer to Unicode length characters. The type for a Unicode character may vary between targets, but it is never the same as the `char` type familiar to all C/C++ programmers. For example, the Unicode character type for Windows, Windows CE and VxWorks is `unsigned short`.

If string pointers or arrays are simply defined to reference `char` types, they cannot hold any Unicode characters. On the other hand, explicitly defining them to reference `unsigned short` types results in code that is not very portable (and will absolutely not run correctly on a more typical non-Unicode target).

When Unicode support is required, it is often times desirable to develop software that will compile with minimum modifications for a Unicode and non-Unicode operating environment (i.e., portable software). A viable solution would allow application code and the Altia API to somehow automatically adjust to the character size based on some compile time macro(s). This is the solution provided by Altia for the Altia API.

5.3 API Header File Supports Portable Application Code

The `altia.h` file that application code includes (with a `#include`) when using the Altia API has an assortment of definitions that automatically adjust based on whether or not

altia.h is compiled for use with the Unicode version of the Altia API. How to compile altia.h for Unicode is described in the upcoming topic [Compiling Application Code to Use the Unicode Version of the Altia API](#).

The first altia.h definition of interest is a typedef for the variable type to use instead of char when sending or receiving strings to/from the Altia API or when manipulating these strings (i.e., getting their length, comparing them, etc.). The typedef defines:

```
AltiaCharType
```

The next definition is for users targeting Windows CE. It is a typedef for the target system's native character size. It is provided because the Windows CE native character type is the size of a Unicode character. All other known target systems have char as the native character type. Application code targeted for Windows CE can use this typedef when referring to string arguments from WinMain(). Application code destined for other targets (including Windows, VxWorks or LINUX) can simply use char for native character strings (e.g., string arguments from WinMain() or main()). This typedef is:

```
AltiaNativeChar
```

The next definition is an extremely useful macro. Any literal string (that is, a string declared by surrounding text between double-quotes as in "Hello World") to be used as an argument to an Altia API function should be wrapped in this macro:

```
AltiaCharType* API_TEXT("string")
```

As an example, to declare a variable powerName that points to the literal string "power_button", the following line of C code might be used:

```
AltiaCharType *powerName = API_TEXT("power_button");
```

As another example, to execute the Altia API function AtSendEvent() for animation "power_led" with a value of 1, the following line of C code might be used:

```
AtSendEvent(API_TEXT("power_led") , (AltiaEventType) 1);
```

When altia.h is NOT compiled for Unicode, the API_TEXT() macro does nothing. Its definition is:

```
#define API_TEXT(x) x
```

When altia.h is compiled for Unicode, the API_TEXT() macro's definition is:

```
#define API_TEXT(x) L ## x
```

The ## is understood by the C preprocessor as a request to concatenate L with x. The C preprocessor takes a source line like:

```
API_TEXT("Hello World")
```

and translates it into the following source line for the C compiler:

```
L"Hello World"
```

When L precedes a literal string, the compiler generates an array of Unicode character length elements. It deposits the appropriate character code into each element of the array (in this case, the codes for 'H' then 'e', etc.) and the last element is set to zero (0) to terminate the string.

Using the `API_TEXT()` macro allows application code to compile without modification for a non-Unicode or Unicode version of the Altia API.

A final set of definitions in `altia.h` help with the manipulation of strings. These definitions are:

```
int API_STRCMP(AltiaCharType*, AltiaCharType*)
int API_STRNCMP(AltiaCharType*, AltiaCharType*, int)
int API_STRLEN(AltiaCharType*)
API_STRCPY(AltiaCharType*, AltiaCharType*)
API_STRNCPY(AltiaCharType*, AltiaCharType*, int)
API_STRCAT(AltiaCharType*, AltiaCharType*)
AltiaCharType* API_STRCHR(AltiaCharType*, AltiaCharType)
double API_STRTOD(AltiaCharType*)
long API_STRTOL(AltiaCharType*)
int API_ISSPACE(AltiaCharType)
int API_ISPRINT(AltiaCharType)
int API_SSCANF(AltiaCharType*, AltiaCharType*, ...)
API_SPRINTF(AltiaCharType*, AltiaCharType*, ...)
```

For example, use `API_STRCMP()` instead of `strcmp()` when application code needs to compile for a Unicode target. For a non-Unicode target, `API_STRCMP()` simply maps to `strcmp()`. For a Unicode target, `API_STRCMP()` maps to the wide string compare function `wcscmp()`.

SPECIAL NOTE ON SUPPORT FOR WIDE STRING MANIPULATION FUNCTIONS

Many target operating systems have built-in support for the wide version of the various string manipulation functions. However, some (such as VxWorks) may not. If a target OS does not have support, compiled code will not link or it will not download without undefined reference errors for `wcslens()`, `wcscmp()`, etc. Please see the later topic [Wide String Manipulation Functions on Targets without Support](#) for a solution.

Here is a classic sample of C code for getting new events from the Altia graphics and processing them. The sample code uses the type definitions and macros just described. This allows it to seamlessly transition between a non-Unicode and Unicode target!

```
#include <altia.h>
AltiaCharType *name;
AltiaEventType value;
AtConnectId altiaId;
altiaId = AtOpenConnection(NULL, NULL, 0, NULL);
AtSelectEvent(altiaId, API_TEXT("power"));
AtSelectEvent(altiaId, API_TEXT("abort"));
...
if (AtPending(altiaId) > 0)
{
    if (AtNextEvent(altiaId, &name, &value) == 0)
    {
        if (API_STRCMP(name, API_TEXT("power")) == 0)
            /* Perform power button action */
        else if (API_STRCMP(name, API_TEXT("abort")) == 0)
            /* Perform power button action */
        }
    }
}
```

5.4 Using AtStartInterface () in a Unicode Version of the API

The argument types for the Unicode version of the Altia API `AtStartInterface()` function are different for interfacing to DeepScreen versus linking with a Unicode Altia API object library to interface to Altia Design or Altia Runtime. If your application code uses this function, please refer to the function declaration for `AtStartInterface()` in the `altia.h` file for details. For a Windows installation of the Altia Design software, this file is `lib/ms32/altia.h` or `libfloat/ms32/altia.h` relative to the Altia Design software installation directory. Open the file in a simple text editor (such as Notepad) and search for occurrences of `AtStartInterface` to locate its function declaration. Especially give attention to the comments under SPECIAL DEEPSCREEN USAGE NOTES.

5.5 Compiling Code to Use the Unicode Version of the Altia API

If the type definitions and macros described above are used consistently in application source code and the `altia.h` file is including (with a `#include`) in the source code, the source code can compile for the Unicode version of the Altia API by defining the following compile time macros:

```
UNICODE
ALTIAUNICODEAPI
```

And to properly adjust for the target specific Unicode character length, `altia.h` defines the type for `AltiaCharType` based on the defining of one, and only one, of the following compile time macros:

```
WCHAR_IS_USHORT
WCHAR_IS_UINT
WCHAR_IS_ULONG
WCHAR_IS_SHORT
WCHAR_IS_LONG
WCHAR_IS_INT
```

For Windows, Windows CE and VxWorks, `WCHAR_IS_USHORT` is the macro to define. On these targets, application code that includes `altia.h`, uses the type definitions and macros described above, and must communicate with the Altia API using Unicode strings, should be compiled with the following compiler flags:

```
-DUNICODE -DALTIAUNICODEAPI -DWCHAR_IS_SHORT
```

5.6 Wide String Manipulation Functions on Targets without Support

If the target system does not provide its own wide string manipulation functions (like `wcslenn()`, `wcscmp()`, etc), the DeepScreen generated code has implementations for these functions (at least for those required by the `API_*` macros presented earlier) (e.g., VxWorks). Normally, the DeepScreen generated code does not compile its versions of these functions under the assumption that they are supplied by the target's LIBC library. To force compilation of the source code for these functions, compile the DeepScreen generated code with the additional compiler flag:

```
-DALTIA_WCHAR_SUPPORT
```

SPECIAL NOTE FOR VxWorks USERS

VxWorks does not have its own support for these functions and so the Altia generated code needs to supply its own implementation of these functions. On Windows and Windows CE, defining this flag is unnecessary because the Windows system libraries already provide these functions.

If a target OS, such as VxWorks, does not implement these functions and the DeepScreen generated code is providing implementations, user application code that takes advantage of one or more of the `API_*` macros must include an additional header file that declares the function prototypes for the functions. This header file is generated as `altia/wCharStrings.h`. User application source code can include it with a line such as:

```
#include <wCharStrings.h>
```

When compiling the user application source code, use the `-I` compiler flag to specify the directory containing `wCharStrings.h`. For example, if the design file for which code is generated resides in `c:\dev\altiaproject`, the DeepScreen generated code for the design file is generated into `c:\dev\altiaproject\altia` and the `-I` compiler flag might look something like:

```
-Ic:\dev\altiaproject\altia
```