Chapter III-16

Text Encodings

Overview

Starting with Igor Pro 7, Igor stores text internally as UTF-8, a Unicode text encoding format. Using Unicode improves Igor's interoperability with other software and gives you easy access to a wide array of characters including Greek, mathematical and special characters.

Previous versions of Igor used non-Unicode text encodings such as MacRoman, Windows-1252, and Shift JIS (Japanese), depending on the operating system you were running. Consequently Igor7 needs to do text encoding conversions when opening files from earlier versions.

If a file contains accented characters, special symbols, and other non-ASCII text, it is not always possible to get this conversion right. You may see incorrect characters or receive “Unicode conversion errors” or other types of errors.

Text encoding is not an issue with files that contain only ASCII characters as these characters are represented using the same codes in UTF-8 as inn other text encodings.

This chapter provides information that a technically-oriented Igor user can use to understand the conversions and to deal with issues that can arise. To understand these issues, it helps to understand what text encodings are and how they are used in Igor, so we start with an overview.

Text Encoding Overview

A text encoding is a mapping from a set of numbers to a set of characters. The terms "text encoding", "character encoding" and "code page" represent the same thing.

In most commonly-used text encodings, text is stored as an array of bytes. For example, the text "ABC" is stored in memory as three bytes with the numeric values 0x41, 0x42 and 0x43. (The 0x notation means the number is expressed in hexadecimal, also called base 16.)

In the 1960's the ASCII text encoding was defined. It specifies the mapping of 128 numbers, from 0x00 to 0x7F, to characters. ASCII defines mappings for:

- Unaccented letters (A-Z and a-z)
- Numerals (0-9)
- Common punctuation marks (comma, period, dash, question mark ...)
- Other printable characters (space, brackets, slash, backslash, ...)
- Control characters (carriage-return, linefeed, tab, ...)

The "ABC" example above uses the ASCII codes for A, B and C.

Over time, other text encodings were created to permit the representation of characters not supported by ASCII such as:

- Accented western letters
- Asian characters

- Mathematical symbols

Hundreds of text encodings were created for encoding characters used in different languages. Even for a given language, different operating systems often adopted different text encodings.

In ASCII, each character is represented by a 7-bit code which is typically stored in a single 8-bit byte with one bit unused. 7 bits can represent 128 different characters. To represent more characters, other text encodings use 8 bits per character, multiple bytes per character, 16-bit codes or 32-bit codes.

We use the term "byte-oriented text encoding" to distinguish text encodings that represent characters using one byte or multiple bytes per character from those that use 16-bit or 32-bit codes.

The characters represented by the ASCII codes 0x00 to 0x7F are called "ASCII characters" with all other characters called "non-ASCII characters". All byte-oriented text encodings that you are likely to encounter are supersets of ASCII.

Multiple-byte text encodings, such as Shift-JIS and UTF-8, typically use one byte for ASCII characters, and for some frequently-used non-ASCII characters, and two or more bytes for other non-ASCII characters.

In order to properly display and process text, a program must convert any text it reads from outside sources (e.g., files) into whatever text encoding the program uses for internal storage if the source text encoding and the internal text encoding are different.

Since all byte-oriented text encodings that you are likely to encounter are supersets of ASCII, ASCII characters do not require conversion when treated as any byte-oriented text encoding. Non-ASCII characters require conversion and problems arise if the conversion is not done or not done correctly.

Text Encodings Commonly Used in Igor

In Igor the most commonly-used text encodings are:

- MacRoman (Macintosh Western European)
- Windows-1252 (Windows Western European)
- Shift JIS (Japanese)
- UTF-8 (Unicode using a byte-oriented encoding form)

Igor6 and earlier versions of Igor stored text in "system text encoding". For western users this means MacRoman on Macintosh and Windows-1252 on Windows. For Japanese users it means Shift JIS on both platforms.

Igor7 uses UTF-8 exclusively.

The system text encoding is determined by the system locale. The system locale determines the text encoding used by non-Unicode programs such as Igor6. It has no effect on Igor7.

On Macintosh you set the system locale through the preferred languages list in the Language & Region preference panel. On Windows you set it through the Region control panel. Most users never have to set the system locale because their systems are initially configured appropriately for them.

Because Igor7 uses UTF-8 internally, when it loads a pre-Igor7 file, it must convert the text from system text encoding to UTF-8. If the file contains ASCII text only this conversion is trivial. But if the file contains non-ASCII characters, such as accented letters, Greek letters, math symbols or Japanese characters, issues may arise, as explained in the following sections.

In order to convert text to UTF-8, Igor must know the text encoding of the file from which it is reading the text. This is tricky because plain text files, including computer program source files, data files, and plain text documentation files, usually provide no information that identifies the text encoding they use. As explained below, Igor7 employs several strategies for determining a file's text encoding so that it can correctly convert the file's text to UTF-8 for internal storage and processing.

Western Text Encodings

MacRoman and Windows-1252 (also called "Windows Latin 1") are used for Western European text and are collectively called "western".

In MacRoman and Windows-1252, the numeric code for each character is stored in a single byte. A byte can represent 256 values, from 0x00 to 0xFF. The 128 values 0x00 through 0x7F represent the same characters as ASCII in both text encodings. The remaining 128 values, from 0x80 to 0xFF, represent different characters in each text encoding. For example the value 0xA5 in MacRoman represents a bullet character while in Windows-1252 it represents a yen symbol.

When Igor6, running on Macintosh, loads an experiment written on Windows, it converts text from Windows-1252 to MacRoman. The opposite occurs if, running on Windows, it loads an experiment written on Macintosh.

When Igor7 loads an Igor6 experiment written on Macintosh, it converts from MacRoman to UTF-8. If the experiment was written on Windows it converts from Windows-1252 to UTF-8.

Asian Text Encodings

Asian languages have thousands of characters so it is not possible to map one character to one byte as in western text encodings. Consequently various multi-byte text encodings were devised for different Asian languages. Such systems are sometimes called MBCS for "multi-byte character system".

Shift JIS is the predominant MBCS text encoding for Japanese. It uses a single byte for all ASCII characters and some Japanese characters and two bytes for all other Japanese characters.

In Shift JIS, the codes from 0x00 to 0x7F have the same meaning as in ASCII except for 0x5C which represents a backslash in ASCII but a yen symbol in Shift JIS. The codes from 0xA1 to 0xDF represent single-byte katakana characters. The codes from 0x81 to 0x9F and 0xE0 to 0xEF are the first bytes of double-byte characters.

Chinese and Korean text encodings also use MBCS.

Unlike the situation for western text, Macintosh and Windows both use the same, or practically the same, text encodings for Japanese. The same is true for Traditional Chinese and Simplified Chinese.

The MacJapanese and Windows-932 text encodings are variants of Shift JIS. In this documentation, "Shift JIS" and "Japanese" mean "Shift JIS or its variants MacJapanese and Windows-932".

When Igor7 loads an Igor6 experiment written in Shift JIS it converts text from Shift JIS to UTF-8.

Unicode

The proliferation of text encodings for different languages and even for the same language on different operating systems complicated the exchange and handling of text. This led to the creation of Unicode - a system for representing virtually all characters in all languages with a single text encoding. The first iteration of Unicode was released in the early 1990's. It was based on 16-bit numbers rather than the 8-bit bytes of previous text encodings.

In 1993 Microsoft released Windows NT, one of the first operating systems to use Unicode internally. Mac OS X, developed in the late 1990's and released in 2001, also uses Unicode internally. Most applications at the time still used system text encoding and the operating systems transparently translated between it and Unicode as necessary.

In the mid-1990's the developers of Unicode realized that 16-bits, which can represent 65,536 distinct numbers, were not enough to represent all characters. They extended Unicode so that it can represent over one million characters by using two 16-bit numbers for some characters. However nearly all of the commonly-used characters can still be represented by a single 16-bit number.

Over time most applications migrated to Unicode for internal storage of text. Igor7 is the first version of Igor that uses Unicode internally.

Unicode Character Encoding Schemes

Unicode maps each character to a unique number between 0 and 0x10FFFF (1,114,111 decimal). Each of these numbers is called a "code point".

There are several ways of storing a code point in memory or in a file. Each of them is called a "character encoding scheme". Each character encoding scheme is based on a "code unit" which may be a byte, a 16-bit value or a 32-bit value.

The most straight-forward character encoding scheme is UTF-32. The code unit is a 32-bit value and each code point is directly represented by a single code unit. UTF-32 is not widely used for historical reasons and because it is not memory-efficient.

UTF-16 is a widely-used character encoding scheme in which the code unit is a 16-bit value and each code point is represented by either one or two code units. Nearly all of the commonly-used characters require just one. Both Mac OS X and Windows use UTF-16 internally.

UTF-8 is a widely-used character encoding scheme in which the code unit is a byte and each code point is represented by one, two, three or four code units. The ASCII characters are represented by a single byte using the same code as in ASCII itself. All other characters are represented by two, three or four bytes. UTF-8 is the predominant character encoding scheme on the Worldwide Web and in email. It is the scheme used internally by Igor7 and later.

Problems Caused By Text Encodings

The use of different text encodings on different operating systems, for different languages, and in different versions of Igor requires that Igor convert between text encodings when reading files. In Igor7 this means converting from the file's text encoding to UTF-8 which Igor7 uses for internal storage. Since ASCII text is the same in UTF-8 as in other byte-oriented text encodings, conversion and the problems it can cause are issues only for files that contain non-ASCII text.

Broadly speaking, when Igor7 opens an Igor6 file, three types of text encoding-related problems can occur:

- Igor's idea of the file's text encoding might be wrong (text encoding misidentification)
- The file may contain invalid byte sequences for its text encoding (invalid text)
- The file may contain a mixture of text encodings (mixed text encodings)

Text Encoding Misidentification Problems

This problem stems from the fact that it is often unclear what text encoding a given file uses. The clearest example of this is the plain text file - the format used for text data files, Igor procedure files and plain text notebooks. A plain text file usually stores just the text and nothing else. This means that there is often nothing in a plain text file that tells Igor what text encoding the file uses.

The problem is similar for plain text items stored within Igor experiment files. These include:

- wave names, wave units, wave notes, wave dimension labels
- text wave contents
- the experiment recreation procedures stored in an experiment file
- the history area contents
- the built-in procedure window's contents
- packed procedure files
- packed plain text notebook files

Igor Pro 6.30 and later, which we will call Igor6.3x for short, store text encoding information in the experiment file for these items but earlier versions of Igor did not.

The text encoding information stored by Igor6.3x and later is not foolproof. Igor6.3x determines the text encoding for a procedure file or a plain text notebook based on the font used to display it. Usually this is correct but it is not guaranteed as the Igor6 user can change a font at any time.

Waves created pre-Igor6.3x have unknown text encodings for their various items.

Igor6.3x determines the text encoding for wave items when the wave is created based on the text encoding used to enter text into the user interface (e.g., into dialogs). This will be the system text encoding except that in IgorJ (the Japanese version of Igor), it is Japanese regardless of the system text encoding. Usually this produces the correct text encoding but it is not guaranteed as it is possible in Igor6 to make waves using any text encoding. For example, when running IgorJ you can switch the command line to an English font and create a wave whose name contains non-ASCII English characters. This name will therefore use MacRoman on Macintosh and Windows-1252 on Windows but IgorJ63x will record the wave name's text encoding as Shift JIS because that is the user interface text encoding. It is similarly possible to create a Japanese wave name when running an English Igor, causing Igor to record the wave name's text encoding as MacRoman or Windows-1252 even though the wave name is really Japanese.

Igor6.3x sets a wave's text encoding information when the wave is created. If a wave was created prior to Igor6.3x and is opened in Igor6.3x, its text encoding is unknown. Saving the experiment in Igor6.3x does not change this - the wave's text encoding is still unknown. So an Igor6.3x experiment can contain a mix of Igor6.3x waves with known text encodings and pre-Igor6.3x waves with unknown text encodings.

The three main causes of text encoding misidentification are:

- The file is a plain text file which contains no text encoding information
- The file is a pre-Igor6.3x experiment file which contains no text encoding information
- The Igor6.3x assumption that the text encoding can be deduced from the text encoding used to enter text into the user interface was incorrect

In order to convert text to UTF-8, Igor7 has to know the text encoding used when the text was created. If it gets this wrong then errors or garbled text will result. Some strategies for fixing such problems are described below.

Invalid Text Problems

A file that contains text in a particular text encoding can be damaged, creating sequences of bytes that are illegal in that text encoding.

In Shift JIS, for example, there are rules about how double-byte characters are formed. The first byte must be in a certain range of values and a second byte must be in a different range of values. If, by error, a line break is inserted between the first and second bytes, the resulting text is not valid as Shift JIS.

Damage like this can happen in Igor6 with long Japanese textboxes. When Igor generates a recreation macro for a graph, it generates TextBox commands for textboxes. If the text is long, Igor breaks it up into an initial TextBox command and subsequent AppendText commands. Igor6 is not smart enough to know that it must keep the two bytes of a double-byte Japanese characters together. This problem is asymptomatic in Igor6 because the two bytes of the double-byte character are reassembled when the macro runs. But it causes an error if you load the Igor6 file into Igor7 because Igor7 must convert the entire file to UTF-8 for internal storage before any reassembly occurs.

There are many other ways to damage a Shift JIS file. Similar damage can occur with other multi-byte text encodings such as UTF-8.

When a file contains invalid text but you suspect that it is mostly OK, you can use the Choose Text Encoding dialog to change the text encoding conversion error handling mode. This allows you to open the file despite the invalid text.

Mixed Text Encoding Problems

In Igor6, you can enter text in different text encodings by merely using different fonts. For example:

1. Run Igor6.3x with an English font controlling the built-in procedure window.
2. Create a graph.
3. Create an annotation using a Japanese font and Japanese characters in the graph.
4. Save the graph as a recreation macro.
5. Save the experiment.

Because an English font controls the built-in procedure window, Igor6.3x thinks that the text encoding for the procedure window is western (MacRoman or Windows-1252) and records this information in the experiment.

When Igor7 opens this experiment, it must convert the procedure window text to UTF-8. Since the Igor6.3x recorded the file's text encoding as western (MacRoman or Windows-1252), Igor7 attempts to convert the text from western to UTF-8. You will wind up with gibberish western characters where the Japanese text should be.

In this example Igor mistook Japanese text for western and the result was gibberish. If Igor mistakes western text for Japanese, you can either wind up with gibberish, or you may get an error, or you may see the Unicode replacement character (a white question mark on a black background) where the misidentified text was.

Fortunately this kind of problem is relatively rare. To fix it you need to manually repair the gibberish text in Igor7.

The Experiment File Text Encoding

Experiments saved by Igor6.3x or later record the "experiment file text encoding". This is the text encoding used by the built-in procedure window at the time the experiment was saved. As explained below, Igor uses the experiment file text encoding as the source text encoding when converting a wave element to UTF-8 if the element's text encoding is unknown.

Prior to version 6.30 Igor did not record the experiment file text encoding so it defaults to unknown when a pre-Igor6.3x experiment is loaded into Igor7. It also defaults to unknown when you first launch Igor7 and when you create a new experiment.

You can display the experiment file text encoding for the currently open experiment in Igor7 by right-clicking the status bar and choosing Show Experiment Info. Igor will display something like "Unknown / 6.22" or "Mac Roman / 6.36". The first item is the experiment file text encoding while the second is the version of Igor that saved the experiment file. This information is usually of interest only to experts investigating text encoding issues.

The Default Text Encoding

Because it is not always possible for Igor7 to correctly determine a file's text encoding, it sometimes needs additional information. You supply this information through the Default Text Encoding submenu in the Text Encoding submenu in the Misc menu. Igor uses your selected default text encoding when it can not otherwise determine the text encoding of a file or wave.

You can display the default text encoding for the currently open experiment in Igor7 by right-clicking the status bar and choosing Show Default Text Encoding. Displaying this information in the status bar is usually of interest only to experts investigating text encoding issues.

Igor experiment files record the platform (Macintosh or Windows) on which they were last saved. In Igor6.3x and later, Igor records text encoding information for most of the items stored in an experiment file. Neither platform information nor text encoding information is stored in standalone plain text files. Consequently determining the text encoding of an experiment file is different from determining the text encoding of a standalone file. This makes Igor's handling of text encodings and the meaning of the default text encoding setting more complicated than we would like.

The Default Text Encoding submenu contains the following items:

```
UTF-8
Western
MacRoman
Windows-1252
Japanese
Simplified Chinese
Traditional Chinese
<Other text encodings>
Override Experiment
```

When you open an experiment in Igor7, Western means MacRoman if the experiment was last saved on Macintosh and Windows-1252 if it was last saved on Windows. When you open a standalone file, Western means MacRoman if you are running on Macintosh and Windows-1252 if you are running on Windows.

The remaining text encoding items, such as MacRoman, Windows-1252 and Japanese, specify the text encoding to use for conversions to UTF-8 without regard to the platform on which the file was last saved or on which you are currently running.

When opening an experiment saved in Igor6.3x or later, Igor normally uses the text encoding information in the experiment file. As explained above under **Text Encoding Misidentification Problems** on page III-413, occasionally this information will be wrong. The Override Experiment item gives you a way to force Igor to use your selected text encoding even when opening experiments that contain text encoding information.

When Igor7 first runs it sets the default text encoding to Western or Japanese based on your system locale. This will give the right behavior when opening Igor6 experiments in most cases. Igor stores the default text encoding in preferences so when your launch Igor again it is restored.

For normal operation, leave the default text encoding setting on Western if you use a Western language or on the appropriate Asian language if you use an Asian language and leave Override Experiment unchecked.

Change these only if a problem arises. For example, if you are running an English system but you are opening a file containing Japanese, or if you are running a Japanese system but you are opening a file containing non-ASCII western text, you may get incorrect text or errors. To fix this, change the settings in this menu and reload the file. After loading the problematic file, reset the menu items for normal operation.

Igor sets the Override Experiment setting to unchecked each time Igor is launched since this is less likely to result in undetected erroneous text conversion than if you turned it on and unintentionally left it turned on.

In addition to affecting text encoding conversion during the loading of files, the default text encoding setting is used as follows:

- When you choose New Experiment, it determines the text encoding of the history area and built-in procedure window. This determines the text encoding used for these items when the experiment is saved to disk.
- When you create a new procedure or notebook window, if a text encoding was not explicitly specified, it determines the text encoding used when the file is saved to disk.
- When you insert text into an existing document using Edit→Insert File, it determines the source text encoding for conversion to UTF-8.
- When Igor loads a wave, it determines the source text encoding for conversion to UTF-8 if the source text encoding is not otherwise specified. This will be the case for waves created before Igor6.3x.
- When the help browser searches procedure files, notebook files and help files, it determines the source text encoding for conversion to UTF-8 if it is not otherwise known.

Plain Text File Text Encodings

Plain text files include procedure files, plain text notebook files, the built-in procedure window text and history text.

Igor converts text that it reads from plain text files to UTF-8 for storage in memory. In order to do this, it needs to know what text encoding to convert from. Most plain text files contain no indication of their text encoding so it can be tricky to determine the source text encoding. If Igor gets the source text encoding wrong, you will get errors or see incorrect characters in the text.

Starting with Igor6.3x, Igor stores text encoding information for plain text files stored in or accessed by experiment files. Experiment files saved before Igor6.3x do not contain text encoding information. Stand-alone plain text files not part of an experiment file, such as Igor6 global procedure files, plain text notebook files and data files, also do not contain text encoding information.

Igor7 looks for the optional byte order mark (see **Byte Order Marks** on page III-420) that some programs write at the start of a Unicode plain text file. If present, the byte order mark unambiguously identifies the file as Unicode and identifies which encoding scheme (UTF-8, UTF-16 or UTF-32) the file uses.

Starting with Igor7, Igor supports a TextEncoding pragma in procedure files that looks like this:

```
#pragma TextEncoding = "<text encoding name>"
```

This is described under **The TextEncoding Pragma** on page IV-51. The TextEncoding pragma allows you to unambiguously specify the text encoding for an Igor procedure file. It is ignored by Igor6.

Igor7 must deal with a variety of circumstances including:

- Standalone files versus experiment files
- Pre-Igor6.3x experiment files versus experiment files from Igor6.3x and later
- Macintosh versus Windows versus Japanese experiment files
- Procedure files with and without a TextEncoding pragma

Because of these complications there is no simple way for Igor7 to determine the source text encoding for any given plain text file. To cope with this situation, Igor7 employs rules described in the next section.

The status bar of a plain text file's window includes a text encoding button which shows you the "file text encoding" - the text encoding that Igor used to convert the file's text to UTF-8. When the file is saved, Igor converts from UTF-8 back to the file text encoding. You can change the file text encoding by clicking the text encoding button to display the Text Encoding dialog.

Determining the Text Encoding for a Plain Text File

This section describes the rules that Igor7 uses to determine the source text encoding for a plain text file.

The rules depend on whether Override Experiment (Misc→Default Text Encoding→Override Experiment) is checked.

Override Experiment Unchecked

If Override Experiment is unchecked, Igor tries the following text encodings, one-after-another, until it succeeds (i.e., it is able to convert to UTF-8 without error), in this order:

```
Byte order mark
TextEncoding pragma if present (for procedure files only)
The specified text encoding (described below)
UTF-8 if the text contains non-ASCII characters
The default text encoding
Choose Text Encoding dialog (described below)
```

Note that a conversion can "succeed" without being correct. For example, Japanese can be successfully interpreted as MacRoman or Windows but you will get the wrong characters.

The "specified text encoding" is the explicit text encoding if present (e.g., specified by the /ENCG flag for the OpenNotebook operation) or the item text encoding (e.g., a text encoding stored in an experiment file for its built-in procedure window text).

If Igor can not otherwise find a valid text source encoding it displays the Choose Text Encoding dialog. This dialog asks you to choose the text encoding to use for the plain text file being opened.

Override Experiment Checked

If the Misc→Default Text Encoding→Override Experiment menu item is checked then the order changes to this:

```
Byte order mark
TextEncoding pragma if present (for procedure files only)
The default text encoding
The specified text encoding (described below)
UTF-8 if the text contains non-ASCII characters
Choose Text Encoding dialog (described below)
```

In this mode the default text encoding, as set in the Misc→Default Text Encoding submenu, is given a higher priority. This should be used only if the normal mode fails to give the correct results. If you turn Override Experiment on, make sure to select the appropriate default text encoding from the submenu. You should turn Override Experiment on only in emergencies and turn it off for normal use.

Potential Problems Determining Plain Text File Text Encodings

The rules listed above have the following potential problems:

TextEncoding pragma (for procedure files only)

The pragma could be wrong. For example, if you set TextEncoding="MacRoman" and then change the file to UTF-8 in an external editor, it will still succeed in Igor but give the wrong characters. This is a programmer error and it is up to you to fix it.

The specified text encoding

There is no specified text encoding for plain text files in a pre-Igor6.3x experiment.

In Igor6.3x and later, Igor includes text encoding information in the experiment restart procedures for each plain text file to be opened. This information is the "specified text encoding".

In Igor6.3x the specified text encoding can be wrong. For example, if the user adds some Japanese characters to a graph annotation, the recreation macro will include Shift-JIS. If the font controlling the procedure window is MacRoman or Windows, the specified text encoding will be MacRoman or Windows and the Japanese characters will show up wrong in Igor7.

UTF-8 if the text contains non-ASCII characters

MacRoman, Windows and ShiftJIS can all masquerade as UTF-8 though this will be rare. By “masquerade” we mean that the non-ASCII bytes in the text are legal as UTF-8 but the text really is encoded using another text encoding.

For a procedure file you can fix this by adding a TextEncoding pragma.

There is currently no fix for plain text notebooks other than setting "Override Experiment" and setting the default text encoding to the correct encoding and reloading or converting the file to UTF-8 with a byte order mark.

The default text encoding

If set to MacRoman or Windows, the default text encoding will always succeed but it might be wrong. For example, Japanese can be successfully interpreted as MacRoman or Windows but you will get the wrong characters.

For a procedure file you can fix this by adding a TextEncoding pragma.

There is currently no fix for plain text notebooks other than setting "Override Experiment" and setting the default text encoding to the correct encoding and reloading or converting the file to UTF-8.

Plain Text Text Encoding Conversion Errors

When Igor7 opens a plain-text file or an Igor6-compatible experiment file, it must convert plain text to UTF-8. In order to do this, it needs to determine the text encoding of the plain text file or the plain text item within an experiment file.

As the previous section briefly outlined, it is possible that Igor might get this wrong. To illustrate the types of problems that can occur in this situation, we will consider the following scenario:

> In Igor6.x you have created a standalone (not part of an experiment) plain text file on Windows with English as the system locale so the text in the file is encoded as Windows-1252. The file contains some non-ASCII characters. You transfer the file to Macintosh. Your default text encoding, as selected in the Default Text Encoding submenu of the Misc menu, is Western. You open the plain text file as a notebook in Igor.

Since the file is is a plain text file, it contains nothing but plain text; it does not contain information indicating the platform of origin or the text encoding. Consequently Igor uses your selected default text encoding - Western. In the case of a plain text file, Western means "MacRoman when running on Macintosh, Windows-1252 when running on Windows", so Igor uses MacRoman to convert the text to UTF-8. This does not cause an error but gives an incorrect translation of the non-ASCII characters in the file.

Unfortunately, Igor has no way to detect this kind of misidentification error. The conversion completes without error but produces some incorrect characters and there is no way for Igor to know that this has happened. This is an example of a conversion that succeeds but is incorrect.

To get the correct characters you must close the file, choose Windows-1252 from the Default Text Encoding menu, and reopen the file. Now Igor will correctly translate the text to UTF-8. You should restore the default text encoding to its normal value for future use.

An alternative solution is to execute OpenNotebook/ENCG=3. This explicitly tells Igor that the file's text encoding is Windows-1252. Igor's text encoding codes are listed under **Text Encoding Names and Codes** on page III-434.

Now let's change the scenario slightly:

> Instead of your default text encoding being set to Windows-1252, it is set to Japanese. You open the standalone plain text file containing Windows western non-ASCII characters as a notebook on either Macintosh or Windows.

Since the file does not contain any text encoding information, Igor tries to convert the text to UTF-8 using default text encoding, Japanese, as the original text encoding. Two things may happen:

- The conversion may succeed. In this case you will see some seemingly random Japanese characters in your notebook because, although the conversion succeeded, it incorrectly interpreted the text.
- The conversion may fail because the file contains byte patterns that are illegal in the Shift JIS text encoding. In this case Igor will display the Choose Text Encoding dialog and you can choose the correct text encoding.

Some Plain Text Files Must Be Saved as UTF-8

When Igor saves a plain text file, it tries to save using the file's original text encoding. Sometimes this is not possible. For example:

You open a file that contains MacRoman text including non-ASCII characters as an Igor procedure file and Igor converts it to UTF-8 for internal storage. Now you add a Japanese character to the procedure file and save it.

The file can not be saved as MacRoman because MacRoman can not represent Japanese characters. In this event, Igor saves the file in UTF-8 and displays an alert informing you of that fact. If you open this file in Igor6, any non-ASCII characters will be incorrect.

If you tell Igor to save a plain text file in a specific text encoding, for example via `SaveNotebook/ENCG=<textEncoding>`, and if it is not possible to save in that text encoding, Igor returns a text encoding conversion error. You can then explicitly tell Igor to save as UTF-8 using `SaveNotebook/ENCG=1`.

History Text Encoding

When an experiment is saved, Igor saves the history text as plain text.

When you create a new experiment, Igor sets the history area text encoding to your chosen default text encoding. When you save the experiment, Igor tries to save the history text using that text encoding. If the history text contains characters that can not be represented in your default text encoding, Igor then saves the history text as UTF-8 and sets the history area text encoding to UTF-8. If you open this experiment in Igor6, any non-ASCII characters in the history, including the bullet characters used to mark commands, will appear as garbage.

The Shift JIS (Japanese) text encoding does not have a character that maps to the bullet character (U+2022). Consequently, if your default text encoding is Shift JIS, when you save an experiment, Igor will be unable to save the history text as Shift JIS and will save it as UTF-8 instead.

Byte Order Marks

A byte order mark, called a BOM for short, is a special character sometimes used in Unicode plain text files to identify the file as Unicode and to identify the Unicode encoding form used by the file. The BOM, if present, must appear at the beginning of the file.

A FAQ about BOMs can be found at http://www.unicode.org/faq/utf_bom.html#BOM.

Most Unicode plain text files omit the BOM. However, including it has the advantage of unambiguously identifying the file's text encoding.

The BOM is the Unicode "zero width no-break space" character. It's code point is U+FEFF which is represented by the following bytes:

UTF-8	0xEF 0xBB 0xBF
UTF-16 Little Endian	0xFF 0xFE
UTF-16 Big Endian	0xFE 0xFF
UTF-32 Little Endian	0x00 0x00 0xFF 0xFE
UTF-32 Big Endian	0xFE 0xFF 0x00 0x00

If a Unicode file contains a BOM, Igor preserves it when saving the file.

By default Igor7 writes a BOM when writing a Unicode plain text file and removes the BOM, if present, when reading a plain text file into memory.

When Igor creates a new plain text file, it sets an internal writeBOM flag for that file to true. If the file is later saved to disk as Unicode and if the writeBOM flag is still set, Igor writes the BOM to the file.

When Igor opens a plain text file, it checks to see if the file contains a BOM. If not it clears the file's writeBOM flag. If the file does contain a BOM, Igor sets the writeBOM flag for the file and removes the BOM from the text as loaded into memory. When Igor writes the text back disk, if the writeBOM flag is set, Igor writes the BOM and then the text.

You can see the state of the writeBOM flag using the File Information dialog which you access via the Notebook or Procedure menu. If the file's text encoding is a form of Unicode and the writeBOM flag is set, the Text Encoding section of the File Information dialog will say "with byte order mark".

You can specify the value of the writeBOM flag for a plain text notebook file using **NewNotebook** with the /ENCG flag.

You can set or clear the writeBOM flag for a plain text notebook or procedure file using the Text Encoding dialog accessed via the Notebook or Procedure menu. The Write Byte Order Mark checkbox is visible only if a Unicode text encoding is selected. You can also set the writeBOM flag for a plain text notebook using the **Notebook** operation, writeBOM keyword.

The built-in procedure window is a special case. Its writeBOM flag defaults to false and it is set to false each time you do New Experiment. We make this exception to allow Igor6 to open an Igor7 experiment whose procedure window text encoding is UTF-8 without generating an error. Igor6 does not know about UTF-8 so non-ASCII characters will be wrong, but at least you will be able to open the experiment.

If you modify a plain text file that is open as a notebook or procedure file using an external editor, Igor reloads the text from the modified file. This sets the internal writeBOM flag for the notebook or procedure file to true if the modified text includes a BOM or false otherwise. This then determines whether Igor writes the BOM if you later save the file. This process of reloading an externally-modified plain text file overwrites the writeBOM flag that you may have set using the Text Encoding dialog or Notebook operation.

BOMs are irrelevant for formatted text notebooks.

Formatted Text Notebook File Text Encodings

Formatted notebook files use an Igor-specific file format. You can open such a file as a formatted notebook or as an Igor help file.

Unlike plain text files, in which all text must be encoded using a single text encoding, Igor6 formatted notebook files can use multiple text encodings in a single file. In Igor6, the text encoding for a given format run is determined by the font controlling that run. For example, a single Igor6 formatted text file can contain western text and Shift JIS (Japanese) text encodings. In fact a single paragraph can contain multiple text encodings.

We use the term MBCS (multi-byte character system) to distinguish these Igor6 formatted text files from Igor7 formatted text files saved using UTF-8.

Igor7 stores text internally (in memory) as UTF-8. Consequently it must convert MBCS text into UTF-8 when opening Igor6 formatted files. When doing this conversion, Igor uses text encoding information written to the file by IP6.3x and later. Pre-IP6.3x formatted text notebook files lack this text encoding information and so may be misinterpreted by Igor7. If you open a formatted text notebook in Igor7 and you get incorrect characters or text encoding conversion errors, try opening and resaving the file in Igor Pro 6.3x, and then reopen in Igor7.

When Igor7 saves a formatted text notebook, it first tries to save it in Igor6-compatible MBCS format. If any format run in the file uses characters that can not be encoded in a single non-Unicode text encoding then it is not possible to save in MBCS format. This would happen, for example, if you mix non-ASCII western characters and Shift JIS characters in a single format run (i.e., controlled by a single font) or if you use a Unicode character that can not be represented in MBCS. In this case, Igor7 saves the file in UTF-8 and dis-

plays an alert explaining the change. If you try to open this file in Igor6.3x, it displays a message in the history area explaining that the file uses UTF-8 so any non-ASCII characters will appear garbled.

In Igor7, to determine how a particular file is saved, click the page icon in the lower-lefthand corner of a formatted notebook or help window. The resulting File Information dialog will say either "MBCS" or "UTF-8" in the Text Encoding area.

If you open an Igor7 UTF-8 formatted text notebook in Igor Pro 6.36 or before and then save the notebook, it is marked as an Igor6 notebook using MBCS even though it really contains UTF-8 text. If you subsequently open the notebook in Igor7, all non-ASCII characters will be incorrectly interpreted as MBCS and will be wrong.

If you open a UTF-8 Igor7 formatted text notebook in Igor Pro 6.37 or a later 6.x version and then save the notebook, it is marked as an Igor7 notebook using UTF-8 text encoding. Igor6 will still display the wrong characters for non-ASCII text, but if you subsequently open the notebook in Igor7, the non-ASCII characters will be correctly interpreted as UTF-8.

Wave Text Encodings

Each wave internally stores several plain text elements. They are:

- wave name
- wave units
- wave note
- wave dimension labels
- text wave content (applies to text waves only)

Each wave stores five settings corresponding to these five plain text elements. The settings record the text encoding used for the corresponding element.

You can inspect the text encodings of the wave elements of a particular wave using the Data Browser. To do so, you must first enable the Text Encoding checkbox in the Info pane of the Data Browser section of the Miscellaneous Settings dialog.

For waves created prior to Igor6.3x, all of the settings are set to 0 which means "unknown". This is because the notion of wave text encodings was added only in Igor Pro 6.30, in anticipation of Igor7 and its use of Unicode as the primary text storage format.

When you make a wave in Igor6.3x the text encoding for each element is set to the system text encoding (typically MacRoman, Windows-1252 or Japanese) except for the text wave content element which is set to "unknown". The reason for treating the text wave content element differntly is because the data may be binary data as explained below under **Text Waves Containing Binary Data** on page III-424.

When you make a wave in Igor7, including when you overwrite an existing wave, the text encoding for each element, including the text wave content element, is set to UTF-8.

These defaults are set when you make or overwrite a wave using Make, Duplicate, LoadWave and any other operation that creates a wave. After creating the wave, you can change them explicitly using **SetWaveTextEncoding**, but only advanced users should do this.

If you overwrite an existing text wave, the text encoding for each element, including the text wave content element, is set to UTF-8, the same as when you first create the wave. However, the wave's text content is not cleared or converted to UTF-8 because it is presumed that you will be completely rewriting the content and, if Igor cleared or converted it on overwriting, it would be a waste of time. If the wave contains non-ASCII text and its previous text content text encoding was not UTF-8, this leaves the wave with invalid text content. This is normally not a problem since you will completely rewrite the text anyway. But if you want to make sure that there is no invalid text, you can clear the wave's text content in the Make/T/O statement. For example:

```
Make/T/O textWave = ""
```

In addition, the text encoding setting for the wave name is set if you rename the wave and the text encoding for the wave note is set if you change the wave note text. However the text encoding for the wave units is not set if you set the units using SetScale, the text encoding for the dimension labels is not set if you set a dimension label using SetDimLabel, and the text encoding for the text content of a text wave is not set if you store text in an element of the wave.

Opening an experiment does not change the text encoding settings for the waves in the experiment. Neither does saving an experiment.

When Igor7 uses one of these wave elements, it must convert the text from its original text encoding into UTF-8. It uses the rules listed in the next section to determine the original text encoding.

Determining the Text Encoding for Wave Elements

This section describes the rules that Igor7 uses to determine the source text encoding for a wave element when loading a wave from an Igor binary wave file or from a packed experiment file. You don't need to know these rules. We present them for technically-oriented users who are interested or who need to understand the details to facilitate troubleshooting.

1. If the element's text encoding is set to UTF-8, Igor uses UTF-8 as the source text encoding.

   Igor7 saves text as UTF-8 if it can not be represented in another text encoding. This rule ensures that Igor7 correctly loads such text.

2. Otherwise if Override Experiment is turned on, Igor uses the selected default text encoding as the source text encoding.

   This provides a way for you to override incorrect text encoding settings when loading an experiment. For example, if you know that an experiment uses Japanese but you get gibberish or Unicode conversion errors when loading it, you can choose Japanese as your default text encoding, select Override Experiment, and reload the experiment. You should restore the default text encoding to its normal value and turn Override Experiment off for future use.

3. Otherwise if the text encoding for the specific element being converted is known, because it was previously set to a value other than "unknown", Igor uses it as the source text encoding.

   This is the effective rule for waves created by Igor6.3x and later.

   This is also the effective rule if you explicitly set a wave's text encoding settings using the **SetWaveTextEncoding** operation.

4. Otherwise if the experiment file text encoding is known, Igor uses it as the source text encoding.

   This rule takes effect for waves created in earlier versions if the experiment was later saved from Igor6.3x or later. Wave text encoding settings would still be unknown but the experiment file text encoding would reflect the text encoding of the experiment's procedure window when it was saved. This will give correct results for most experiments.

5. Otherwise Igor uses the selected default text encoding as the source text encoding.

   This rule takes effect for experiments saved before Igor6.3x in which both the experiment file text encoding and the element-specific text encodings are unknown. This is the effective rule for experiments of this vintage.

Wave Text Encoding Problems

This section describes the problems that may occur when loading a wave from an Igor binary wave file or from a packed experiment file.

If Igor7 is not able to correctly determine a wave element's text encoding and if the element contains non-ASCII text you may see gibberish text or Igor may generate a text encoding conversion error.

If Igor7 is not able to correctly determine the text encoding of a wave name you will see gibberish for the name and Igor will be unable to find the wave when it is referenced from procedures such as experiment recreation procedures. This may cause errors when the experiment is loaded.

Igor's handling of invalid text in tables is described under **Editing Invalid Text** on page II-194.

The **SetWaveTextEncoding** operation allows you to set the text encodings for one or more waves or for all waves in the experiment. This is especially useful for pre-IP6.3x experiments. You can run it in Igor6.3x as well as Igor7. It is discussed below under **Manually Setting Wave Text Encodings** on page III-427.

If you get text encoding conversion errors when opening an experiment you can try the following:

- Choose a different text encoding from the Default Text Encoding menu and reopen the experiment
- Select Override Experiment in the Default Text Encoding menu so that it is checked and reopen the experiment
- Manually fix the errors after opening the experiment in Igor7
- Use the **SetWaveTextEncoding** operation to specify the text encodings used by waves
- For pre-IP63x experiments, open the experiment in Igor6.3x, resave it, and reopen it in Igor7

After attempting to fix a text encoding problem you should restore your default text encoding to the most reasonable setting for your situation and turn Override Experiment off.

If you are unable to solve the problem you can send the experiment to WaveMetrics support with the following information:

- What operating system you are running
- What Igor version you are running
- What incorrect output you are seeing and what you expect to see

LoadWave Text Encodings for Igor Binary Wave Files

The rules that LoadWave uses to determine the source text encoding of an Igor binary wave file are the same as the rules described under **Determining the Text Encoding for Wave Elements** on page III-423. The LoadWave /ENCG flag is ignored when loading an Igor binary wave file.

LoadWave Text Encodings for Plain Text Files

This section describes the rules that the LoadWave operation uses to determine the source text encoding when loading a plain text file. This includes general text, delimited text and Igor text files but not Igor binary wave files.

The rules that LoadWave uses to determine the source text encoding of a plain text file are the same as the rules described under **Determining the Text Encoding for a Plain Text File** on page III-417. The "specified text encoding", which is one of the factors used by the rules, is unknown unless you use the /ENCG flag to tell LoadWave the file's text encoding.

For further details see *LoadWave Text Encoding Issues* in the reference documentation for **LoadWave**.

Text Waves Containing Binary Data

You can set a text wave's content text encoding to the special value 255 using **SetWaveTextEncoding**. This marks a text wave as really containing binary data, not text.

Text is data that is represented by numeric codes which map to characters using some recognized text encoding such as MacRoman, Windows-1252, Shift JIS or UTF-8. By contrast, binary data does not follow any recognized text encoding and generally does not map to characters. Whereas text data is typically

intended to be readable by humans using a text editor, binary data is intended to be processed only by programs.

You can mark any text wave element as binary but it is normally done only for text wave content because it is rare to store binary data in wave units, wave notes or wave dimension labels.

Igor text waves were designed to store text but are able to store any collection of bytes.

While a numeric wave stores a fixed number of bytes in each element, a text wave has the ability to store a different number of bytes in each point. For example:

```
Make /O /T /N=2 twave0
twave0[0] = "Hello"                    // 5 bytes
twave0[1] = "Goodbye"                  // 7 bytes
```

This capability makes a text wave a convenient way to store a collection of binary arrays of different length. For example, you could store the contents of a different downloaded file in each element of a text wave:

```
Make /O /T /N=2 twave0
twave0[0] = FetchURL("http://www.wavemetrics.net/images/tbg.gif")
twave0[1] = FetchURL("http://www.wavemetrics.net/images/mbg.gif")
```

While most text waves contain text data, clever programmers sometimes use text waves to store binary data. Since binary data does not follow any recognized text encoding, it must not be treated as text. For example, you can convert a text wave from Shift JIS to UTF-8 and preserve the meaning of the text; you are merely changing how the characters are encoded. By contrast, if you attempt to convert binary data which you have mistaken for text, you turn your data into garbage.

The **SetWaveTextEncoding** operation can convert text waves from one text encoding to another but you never want to convert binary text waves. SetWaveTextEncoding conveniently skips text waves marked as binary when doing a conversion but it is up to you to mark such waves appropriately. Here is an example that illustrates the utility of marking a text wave as containing binary data:

```
// Mark contents as binary
SetWaveTextEncoding 255, 16, root:MyBinaryTextWave
// Convert contents to UTF-8 skipping binary waves
SetWaveTextEncoding /DF={root:,1} /CONV=1 1, 16
```

The first command marks a particular wave's content (16) as containing binary (255) as opposed to text. The second command converts all text wave text to UTF-8 (1) except that it automatically skips waves marked as binary.

This is fine but it would be tedious to identify and mark each text wave containing binary data. So SetWaveTextEncoding provides a way to do it automatically:

```
// Mark all binary waves as binary
SetWaveTextEncoding /BINA=1 /DF={root:,1} 255, 16
// Convert to UTF-8 skipping binary waves
SetWaveTextEncoding /DF={root:,1} /CONV=1 1, 16
```

The first command goes through each wave in the experiment. If it is a text wave, it examines its contents. If the wave contains control characters (codes less than 0x20) other than carriage-return (0x0D), linefeed (0x0A) and tab (0x09), it marks the wave as containing binary. Then the second command converts all non-binary text waves to UTF-8. These commands can be combined into one:

```
// Mark as binary then convert non-binary to UTF-8
SetWaveTextEncoding /BINA=1 /DF={root:,1} /CONV=1 1, 16
```

NOTE: The heuristic used by the /BINA flag is not foolproof. While most binary data will contain 0x00 bytes or other bytes that flag it as binary, it is possible to have binary data that contains no such bytes. In that case, the first command would fail to mark the text waves as containing binary and the second command would

either return an error or clobber the binary data. For this reason, it is imperative that you back up any data before doing text encoding conversions.

PROGRAMMER'S NOTE: If you are a programmer and you create text waves to store binary data, be sure to mark them as binary. For example:

```
Wave/T bw = <path to some text wave used to contain binary data>
SetWaveTextEncoding 255, 16, bw        // Mark text wave as binary
```

Doing this will ensure that your binary data is not "converted" to some text encoding, thereby clobbering it.

Handling Text Waves Containing Binary Data

Usually you use a text wave containing binary data only as a container to be accessed by procedures that understand how to interpret the contents. You normally should not treat it as text. For example, you should not print its contents to the history because this treat the wave's binary contents as text. If you do treat it as text you are likely to get gibberish or a Unicode conversion error.

You can view a text wave marked as binary in a table and you can cut and paste its cells but you can not edit the binary data in the table's entry line. You can dump the contents of a cell as hex to the history area by pressing Cmd (*Macintosh*) or Ctrl (*Windows*) and double-clicking the cell.

Igor7 normally does automatic text encoding conversion when fetching data from a text wave or storing data into a text wave. For example:

```
// Make a wave to contain text in UTF-8 text encoding
Make /O /T UTF8 = {"<some Japanese text>"}      // UTF-8 by default

// Make a wave to contain text in Shift JIS text encoding
Make /O /T /N=1 ShiftJIS                       // UTF-8 by default
SetWaveTextEncoding 4, 16, ShiftJIS            // Now Shift JIS

ShiftJIS = UTF8
```

The last statement triggers a fetch from the wave UTF8 and a store into the wave ShiftJIS. A fetch converts a text wave's data to UTF-8. In this case it is already UTF-8 so no conversion is done. A store converts the data to the text encoding of the destination wave - Shift JIS in this case. You wind up with the same characters in ShiftJIS and in UTF8 but not the same raw bytes because they are governed by different text encodings.

Igor7 does not do text encoding conversion when fetching from a wave or storing into a wave if the wave is marked as binary. For example:

```
// Make a wave to contain binary data
Make /O /T /N=1 Binary                         // UTF-8 by default
SetWaveTextEncoding 255, 16, Binary            // Now binary
Binary = UTF8
```

The last statement's storing action does not do text encoding conversion because the destination wave is marked as binary. Thus Binary will wind up with the same bytes as in UTF8.

Now consider this:

```
Binary = ShiftJIS
```

Once again the last statement's storing action does not do text encoding conversion because the destination wave is marked as binary. You might think that Binary would wind up with the same bytes as in ShiftJIS but you would be wrong. The reason is that the fecthing action triggered by evaluating the righthand side of the assignment statement converts the Shift JIS text in ShiftJIS to Igor's internal standard, UTF-8. Consequently the same bytes wind up being stored in Binary after each of the two previous statements.

The moral of the story is that, if you want to transfer binary data between text waves, make sure that both waves are marked as binary. It will also work if the source wave is marked as UTF-8 since the fetch conversion does nothing if the source is UTF-8.

Manually Setting Wave Text Encodings

NOTE: Back up your experiment before fiddling with text encoding settings.

With the concepts of wave plain text elements, their text encoding settings, and text waves containing binary data in mind, we can now consider a general approach for setting wave text encodings. This is a task for advanced users only.

As a test case, we will assume that you created an experiment using Igor Pro 6.22 on Windows with English as the system locale. Consequently the experiment contains waves with all text encoding settings set to unknown. You then opened the experiment in Igor Pro 6.30, again on Windows with English as the system locale, and created additional waves. These added waves have text encoding settings set to Windows-1252 except for the text wave content element which is set to unknown. Finally, somewhere along the line, some text waves were created to store binary data, possibly by a package that you use and without your knowledge.

We now open the experiment in Igor7 and try to set all of the text encoding settings to correct known values using the SetWaveTextEncoding operation. The magic numbers used in the following commands are detailed below under **Text Encoding Names and Codes** on page III-434 and **SetWaveTextEncoding**.

```
// Mark any text waves containing binary data as such.
// 255 means "binary". 16 means "text wave content".
// /DF={root:,1} means "all waves in all data folders".
// /BINA=1 means "automatically mark text waves containing binary data".
SetWaveTextEncoding /DF={root:,1} /BINA=1 255, 16

// Mark any wave text elements currently set to unknown as Windows-1252.
// 0 means "unknown". 3 means Windows-1252. 31 means "all wave elements".
// /ONLY=0 means apply the command only to wave elements currently marked as
unknown.
SetWaveTextEncoding /DF={root:,1} /ONLY=0 3, 31
```

Now all elements of all waves are set to Windows-1252 except for the content element of text waves containing binary data which are so marked. In other words, there are no unknown text encodings and all elements are correctly marked. Marking all waves correctly ensures that Igor7 can correctly convert the wave plain text elements to UTF-8 for internal use.

If you were starting from a MacRoman experiment (Macintosh western text), you would use 2 instead of 3. If you were starting from a Japanese experiment (Shift JIS), you would use 4 instead of 3.

NOTE: The heuristic used by the /BINA flag is not foolproof. See **Text Waves Containing Binary Data** on page III-424 for details.

At this point you may want to convert your waves from Windows-1252 to UTF-8. This provides two advantages. First, Igor7 will not need to convert to UTF-8 when fetching the text waves' contents since the waves will already be UTF-8. Second, you will be able to use a wider repertoire of characters. The disadvantage is that you will get gibberish for non-ASCII characters if you open the experiment in any pre-Igor7 version. Assuming that you do want to convert to UTF-8, you would execute this:

```
// 1 means UTF-8. 31 means "all wave elements".
// /CONV means "convert text encoding". It automatically skips text wave
// content marked as binary.
SetWaveTextEncoding /DF={root:,1} /CONV=1 1, 31
```

Data Folder Name Text Encodings

Igor Pro 7.00 through 7.01 were unable to correctly load non-ASCII data folder names from Igor6 packed experiment files. Starting with version 7.02, in most cases, Igor correctly loads such data folder names.

If Igor mangles a non-ASCII data folder name when loading an Igor6 experiment, try choosing the correct text encoding from the Misc->Text Encoding->Default Text Encoding menu and turn Misc->Text Encoding->Default Text Encoding->Override Experiment on. Remember to restore these settings to their original settings after loading the experiment.

Starting with Igor Pro 7.02, Igor writes non-ASCII data folder names to packed experiment files using the text encoding of the built-in procedure window if possible. This will be Igor6-compatible if that text encoding is MacRoman on Macintosh or Windows-1252 on Windows. Igor Pro 7.00 through 7.01 expect UTF-8 and will therefore misinterpret such data folder names unless the text encoding of the built-in procedure window is UTF-8.

String Variable Text Encodings

Unlike the plain text elements of waves, there is no text encoding setting for string variables. Consequently Igor7 treats a string variable as if it contains UTF-8 text when printing it to the history area or when displaying it in an annotation or control panel or otherwise treating it as text.

If you have string variables in Igor6 experiments that contain non-ASCII characters, they will be misinterpreted by Igor7. This may cause gibberish text or it may cause a Unicode conversion error. In either case you must manually fix the problem by storing valid UTF-8 text in the string variable.

For a local string variable, you can achieve this by assigning a new value to the string variable or by converting its contents from its original text encoding to UTF-8 using the **ConvertTextEncoding** function.

For a global string variable, you can achieve this by converting its contents from its original text encoding to UTF-8 using the **ConvertGlobalStringTextEncoding** operation. You need to know the original text encoding. It will typically be MacRoman if the Igor6 experiment was created on Macintosh, Windows-1252 if it was created on Windows, or Shift JIS if it was created using a Japanese version of Igor.

ConvertGlobalStringTextEncoding can convert all global string variables in the current experiment at once. Choose Misc→Text Encoding→Convert Global String Text Encoding to generate a ConvertGlobalStringTextEncoding command.

If you don't print or display a string variable or compare it to other text then its text encoding does not matter. For example, if you have a string variable that contains binary data and it is treated by your procedures as binary data, then everything will work out.

String Variable Text Encoding Error Example

This section gives a concrete example of a text encoding error which may help you to understand text encoding issues. As you read the example, keep these thoughts in mind:

- A text wave stores the raw bytes representing its text and an explicit setting that tells Igor the text encoding of the raw bytes. When you access an element of a text wave, Igor converts it to UTF-8 if it is not already stored as UTF-8 and is not marked as binary. During this conversion Igor uses the wave's explicit text encoding setting.
- String variables and other string expressions do not have text encoding settings and are always assumed to contain text encoded as UTF-8.

This example shows how you can create a text encoding problem and how you can avoid it.

```
Function TextEncodingErrorExample()
   // Make a text wave. Its text encoding defaults to UTF-8.
   Make /O/T/N=1 textWave0
```

```
   // Create a string variable containing non-ASCII text (the character ¹).
   // Igor converts literal text ("¹ != 3" in this case) to UTF-8
   // and stores it in the string variable.
   String text = "¹ != 3"                    // Stored as UTF-8

   // Convert the string variable contents to MacRoman.
   // Since the string has no text encoding setting, Igor still assumes it
   // contains UTF-8 but we know it really contains MacRoman.
   Variable textEncodingUTF8 = TextEncodingCode("UTF-8")
   Variable textEncodingMacRoman = TextEncodingCode("MacRoman")
   text = ConvertTextEncoding(text,textEncodingUTF8,textEncodingMacRoman,1,0)

   // Store the MacRoman text in the text wave. This creates bad data
   // because Igor assumes you are storing UTF-8 text in a UTF-8 text wave
   // when in fact you are storing MacRoman into a UTF-8 text wave.
   textWave0 = text                          // Store incorrect data

   // Print - an incorrect character will be printed because the wave's
   // text encoding setting is incorrect for the raw data bytes stored in it.
   Print textWave0[0]

   // Here is one way to fix the problem - convert the string variable to UTF-8.
   String textUTF8 = ConvertTextEncoding(text, textEncodingMacRoman,
                                         textEncodingUTF8, 1, 0)
   textWave0 = textUTF8                      // Store correct data

   // Print again - this time it prints correctly because the wave's
   // raw text bytes are consistent with the wave text encoding setting.
   Print textWave0[0]

   // Create the problem again so we can see another way to fix it.
   textWave0 = text                          // Store incorrect data

   // Here is another way to fix the problem - by correcting Igor's idea
   // of what text encoding the wave is using.
   // Tell Igor that the text wave stores text encoded as MacRoman.
   SetWaveTextEncoding textEncodingMacRoman, 16, textWave0

   // Print again - this time it prints correctly because Igor knows
   // that the wave contains MacRoman and so converts from MacRoman
   // to UTF-8 before sending the text to the history.
   Print textWave0[0]
End
```

Text Encoding Programming Issues

Igor6 and earlier versions of Igor stored text in "system text encoding". For western users this means MacRoman on Macintosh and Windows-1252 on Windows. For Japanese users it means Shift JIS on both platforms.

Igor7 uses UTF-8, a form of Unicode, exclusively.

If you are an Igor programmer attempting to support both Igor6 and Igor7, and if you or your users use non-ASCII text, the introduction of Unicode in Igor7 entails compatibility challenges.

If you fall in this category, we highly recommend that you abandon the idea of supporting Igor6 and Igor7 with the same procedure files. Instead, freeze your Igor6 code, and do all new development in the Igor7 fork. Here are the reasons for not attempting to support Igor6 and Igor7 with the same code base:

1. You will be unable to use new Igor7 features without using ifdefs and other workarounds.

2. The use of ifdefs and workarounds will clutter your code and introduce bugs in formerly working Igor6 code.

3. You will have to deal with tricky text encoding issues, as explained in the next section.

4. Eventually Igor6 will be ancient history.

The alternative to supporting both Igor6 and Igor7 with the same procedure files is to create two versions of your procedures, one version for Igor6 and another for Igor7 and beyond. The benefits of this approach are:

1. You avoid cluttering your Igor6 code or introducing new bugs in it.

2. It will be much easier to write Igor7 code and it will be cleaner.

3. Your Igor7 code can use the full range of Unicode characters.

The costs of this approach are:

1. If you fix a bug in the Igor6 version of your code, you will have to fix it in the Igor7 version also.

2. New features of your code will be unavailable to Igor6 users.

The next section gives a taste of the text encoding issues raised by the use of Unicode in Igor7.

Literal Strings in Igor Procedures

Consider this function, which prints a temperature with a degree sign in Igor's history area:

```
Function PrintTemperature(temperature)
   Variable temperature

   // The character after %g is the degree sign
   Printf "The temperature is %g°\r", temperature
End
```

The degree sign is a non-ASCII character. It is encoded as follows:

MacRoman	0xA1
Windows-1252	0xB0
Shift JIS (Japanese)	0x81 0x8B
UTF-8	0xC2 0xB0

To simplify the discussion, we will ignore the issue of Japanese.

If you created the procedure file in Igor6 on Macintosh, it would work correctly in Igor6 on Macintosh but would print the wrong character in Igor6 on Windows. If you opened the procedure file in Igor7 with "Western" selected in Misc→Text Encoding→Default Text Encoding, it would work correctly on Macintosh but would print the wrong character on Windows. If you added this to the procedure file:

```
#pragma TextEncoding = "MacRoman"
```

it would work correctly in Igor7 on both Macintosh and Windows.

Literal Strings in Igor6/Igor7 Procedures

If you want it to work in Igor6 and Igor7, on Macintosh and Windows, and irrespective of the text encoding of the procedure file and the Misc→Text Encoding→Default Text Encoding setting, you will need to do this:

```
#if IgorVersion() >= 7.00
   static StrConstant kDegreeSignCharacter = "\xC2\xB0"
```

```
#else
   #ifdef MACINTOSH
      static StrConstant kDegreeSignCharacter = "\241"
   #else
      static StrConstant kDegreeSignCharacter = "\260"
   #endif
#endif

Function PrintTemperature(temperature)
   Variable temperature

   // The character after %g is the degree sign
   Printf "The temperature is %g%s\r", temperature, kDegreeSignCharacter
End
```

This code uses escape sequences instead of literal characters to keep non-ASCII characters out of the procedure file. This allows it to work regardless of the text encoding of the procedure file in which it appears. For Igor7, it uses the `\xXX` hexadecimal escape sequence. Igor6 does not support `\xXX` so it uses octal (`\OOO`) instead.

The section **Determining the Encodings of a Particular Character** on page III-432 shows how we determined the hex and octal codes.

The reason for making the StrConstants static is that other programmers may include the same code, causing a compile error if a user uses both sets of procedures.

The string constant kDegreeSignCharacter produces the correct string in Igor6 on Macintosh and Windows, and in Igor7 on Macintosh and Windows.

This will not print the right character for an Igor6 user using a Japanese font for the history area. Since you can not know the user's history area font, there is no way to cope with that.

If you have code that assumes that degree sign is one byte, you will need to modify it, since it is two bytes in UTF-8.

If the degree sign is the only non-ASCII character that you need to represent in Igor6 and Igor7, this problem is manageable. If not, you will need similar code for other characters. The additional complexity and kludginess will get out of hand. That's why we recommend against supporting Igor6 and Igor7 with the same set of procedures.

Literal Strings in Igor7-only Code

If your procedure file requires Igor7 or later, you can write the PrintTemperature function in the obvious way:

```
#pragma TextEncoding = "UTF-8"
#pragma IgorVersion = 7.00

Function PrintTemperature(temperature)
   Variable temperature

   // The character after %g is the degree sign
   Printf "The temperature is %g°\r", temperature
End
```

This will work in Igor7 or later on Macintosh and Windows, and without regard to the user's history font or the Misc→Text Encoding→Default Text Encoding setting.

Unlike the previous approaches, this technique allows you to use any Unicode character, not just the small subset available in MacRoman or Windows-1252.

Determining the Encodings of a Particular Character

This section shows a method for determining how a given character is encoded in MacRoman, Windows-1252, and UTF-8. The commands must be executed in Igor7 because they use the ConvertTextEncoding function which does not exist in Igor6.

```
// Print MacRoman code for degree sign as octal - Prints 241
Printf "%03o\r", char2num(ConvertTextEncoding("°", 1, 2, 1, 0)) & 0xFF

// Print Windows-1252 code for degree sign as octal - Prints 260
Printf "%03o\r", char2num(ConvertTextEncoding("°" , 1, 3, 1, 0)) & 0xFF

// Print number of bytes of UTF-8 degree sign character - Prints 2
Printf "%d\r", strlen("°")

// Print first byte of UTF-8 for degree sign as hex - Prints C2
Printf "%02X\r", char2num("°"[0]) & 0xFF

// Print second byte of UTF-8 for degree sign as hex - Prints B0
Printf "%02X\r", char2num("°"[1]) & 0xFF
```

Using the printed information, we created the following Igor6/Igor7 code. It uses a hex escape sequence in Igor7 and an octal escape sequence in Igor6 because Igor6 does not support hex escape sequences.

```
#if IgorVersion() >= 7.00
   static StrConstant kDegreeSignCharacter = "\xC2\xB0"
#else
   #ifdef MACINTOSH
      static StrConstant kDegreeSignCharacter = "\241"
   #else
      static StrConstant kDegreeSignCharacter = "\260"
   #endif
#endif

Function PrintTemperature(temperature)
   Variable temperature

   // The character after %g is the degree sign
   Printf "The temperature is %g%s\r", temperature, kDegreeSignCharacter
End
```

This web page is a useful resource for comparing MacRoman and Windows-1252 text encodings: https://kb.iu.edu/d/aesh

The MacRoman column is labeled "Mac OS" and the Windows-1252 column is labeled "Windows". The Latin1 column is a subset of Windows-1252.

You can see from the table that there are many characters that are available in MacRoman but not in Windows-1252 (e.g., Greek small letter pi) and vice-versa (e.g., multiplication sign). This is one of the reasons for switching to Unicode.

The left column of the table shows UTF-16 character codes. The table does not show UTF-8. You can convert a UTF-16 character code to UTF-8 text in Igor7 like this:

```
String piCharacter = "\u03C0"
```

For details on "\u", see **Escape Sequences in Strings** on page IV-13.

Other Text Encodings Issues

This section discusses miscellaneous issues relating to text encodings.

Characters Versus Bytes

In olden times, each character was represented in memory by a single byte. For example, in ASCII, A was represented by 0x41 (0x means hexadecimal) and B was represented by 0x42. Life was simple.

A single byte can represent 256 unique values so it was possible to represent 256 unique characters using a single byte. This was enough for western languages.

There came a time when people wanted to represent Asian text in computers. There are far more than 256 Asian characters, so multi-byte character sets (MBCS), such as Shift JIS, were invented. In MBCS, the ASCII characters are represented by one byte but most Asian characters are represented by two or more bytes. At this point, the equivalence of character and byte was no more and it became important to distinguish the two.

For example, in Igor6, the documentation for strlen said that it returns the number of characters in a string. This was true for western text but not for Asian text. The documentation was changed in Igor7 to say that strlen returns the number of bytes in a string.

In Igor7, Igor was changed to store text internally as UTF-8, a byte-oriented Unicode encoding form. In UTF-8, ASCII characters are represented by the same single-byte codes as in ASCII, thus providing complete backward compatibility for ASCII text. However, all non-ASCII characters, including accented characters, Greek letters, math symbols and special symbols, are represented by multiple bytes. A given character can consist of 1, 2, 3 or 4 bytes. The payback for this complexity is that UTF-8 can represent nearly every character of nearly every language.

In Igor6, this command prints 1 but in Igor7, it prints 2:

```
Print strlen("µ")
```

This distinction is immaterial when you treat a string as a single entity, which is most of the time. For example, this code works correctly regardless of how characters are represented:

```
String units = "µs"
Printf "The units are %s\r", units
```

It is when you tinker with the contents of a string that it makes a difference. This works correctly in Igor6 and incorrectly in Igor7:

```
String units = "µs"
Printf "The unit prefix is %s\r", units[0]
```

This fails in Igor7, and prints a missing character symbol, because the expression "units[0]" returns a single byte, but in UTF-8, "µ" is represented by two bytes. Therefore "units[0]" returns the first byte of a two-byte character and that is an invalid character in UTF-8.

If you are tinkering with the contents of a string, and if the string may contain non-ASCII text, you need to be clear when you want to go byte-by-byte and when you want to go character-by-character. To go byte-by-byte, you merely use sequential indices. At present, there is no built-in support in Igor for going character-by-character. See **Character-by-Character Operations** on page IV-162 for an example of stepping through characters using user-defined functions.

Automatic Text Encoding Detection

Techniques exist for attempting to determine a file's text encoding from the codes it contains. Such techniques are unreliable, especially with the kind of text commonly used in Igor. Consequently Igor does not use them.

Much of Igor plain text is procedure text. Procedure text is mostly ASCII, sometimes with a smattering of non-ASCII characters. The non-ASCII characters may be, for example, MacRoman, Windows-1252, or Japanese.

There is no reliable way to distinguish MacRoman from Windows-1252 or to reliably distinguish a procedure file that contains a smattering of non-ASCII western text from one that contains a smattering of Japanese.

For example, consider an Igor procedure file that contains just one Japanese two-byte character encoded in Shift JIS by the bytes 0x95 and 0x61. This can be interpreted as:

Shift JIS:	CJK UNIFIED IDEOGRAPH-75C5
MacRoman:	LATIN SMALL LETTER I WITH DIERESIS, LOWERCASE LETTER A
Windows-1252:	BULLET, LOWERCASE LETTER A

All three of these interpretations are possible and choosing among them is guesswork so Igor does not attempt to do it.

Shift JIS Backslash Issue

Because Igor7 internally process all text as UTF-8, it must convert text that it reads from files that use other encodings. A special issue arises with Japanese text stored in Shift JIS format.

In Shift JIS, the single-byte code 0x5C is hijacked to represent the half-width yen symbol. In ASCII, 0x5C represents the backslash character. When Shift JIS text is converted to UTF-8, Igor's conversion software leaves 0x5C unchanged. Thus its interpretation changes from a half-width yen symbol to a backslash symbol.

In most cases this is the desired behavior because the half-width yen symbol is used in Japanese like the backslash in ASCII - to separate elements of Windows file system paths and to introduce escape sequences in literal strings.

If you are using a half-width yen symbol as a currency symbol then this conversion will be wrong and you will have to manually convert it to a half-width yen symbol by editing the text.

In UTF-8, the half-width yen currency symbol is represented by the code units 0xC2 and 0xA5. It can be entered in a literal text string as "\xC2\xA5".

Text Encoding Names and Codes

Igor operations and functions such as **ConvertTextEncoding**, **SaveNotebook**, **WaveTextEncoding** and **SetWaveTextEncoding** accept text encoding codes as parameters or return them as results.

The functions **TextEncodingName** and **TextEncodingCode** provide conversion between names and codes. For most Igor programming purpose, you need the code, not the name.

For each text encoding supported by Igor there is one text encoding code. This code may correspond to one or more text encoding names. For example, code 2 is the Macintosh Roman text encoding and corresponds to the text encoding names "macintosh", "MacRoman" and "x-macroman".

Because spelling of text encoding names is inconsistent in practice, Igor recognizes variant spellings such as "Shift JIS", "ShiftJIS", "Shift_JIS" and "Shift-JIS". When comparing text encoding names, Igor ignores all non-alphanumeric characters. It also ignores leading zeros in numbers embedded in text encoding names so that "ISO-8859-1" and "ISO-8859-01" refer to the same text encoding. It uses a case-insensitive comparison.

Some of the entries in the table below are marked NOT SUPPORTED. These are not supported because the ICU (International Components for Unicode) library, which Igor uses for text encoding conversions, does

not support it. "Not supported" means that the TextEncodingName and TextEncodingCode functions do not recognize these text encodings and Igor can not convert to them or from them.

Text Encoding Name	Code	Notes
None	0	Means the text encoding is unknown
UTF-8	1	Unicode UTF-8
macintosh, MacRoman, x-macroman	2	Macintosh Western European
Windows-1252	3	Windows Western European
Shift_JIS	4	Predominant Japanese text encoding
MacJapanese, x-mac-japanese	4	Virtually the same as Shift_JIS
Windows-932	4	Virtually the same as Shift_JIS
EUC-JP	5	Japanese, typically used on Unix
Big5	20	Traditional Chinese
x-mac-chinesetrad	20	Virtually the same as Big5
Windows-950	20	Virtually the same as Big5
EUC-CN	21	Simplified Chinese
x-mac-chinesesimp	21	Simplified Chinese
Windows-936	21	Simplified Chinese
ISO-2022-CN	22	Simplified Chinese
GB18030	23	Official text encoding of the PRC. Encompasses Traditional Chinese and Simplified Chinese. Compatible with Windows-936.
EUC-KR, x-mac-korean	40	Macintosh Korean
Windows-949, ks_c_5601-1987	41	Windows Korean
ISO-2022-KR	42	Korean text encoding not used on Macintosh or Windows
x-mac-arabic	50	Macintosh Arabic. NOT SUPPORTED.
Windows-1256	51	Windows Arabic
x-mac-hebrew	55	Macintosh Hebrew
Windows-1255	56	Windows Hebrew
x-mac-greek	60	Macintosh Greek
Windows-1253	61	Windows Greek
x-mac-cyrillic	65	Macintosh Cyrillic
Windows-1251	66	Windows Cyrillic
x-mac-thai	70	Macintosh Thai. NOT SUPPORTED.
Windows-874	71	Windows Thai
x-mac-ce	80	Macintosh Central European
Windows-1250	81	Windows Central European
x-mac-turkish	90	Macintosh Turkish
Windows-1254	91	Windows Turkish

Text Encoding Name	Code	Notes
UTF-16BE	100	UTF-16, big-endian
UTF-16LE	101	UTF-16, little-endian
UTF-32BE	102	UTF-32, big-endian
UTF-32LE	103	UTF-32, little-endian
ISO-8859-1, Latin1	120	ISO standard western European
Symbol	150	Used by Symbol font
Binary	255	Indicates data is really binary, not text

Binary is not a real text encoding. Rather it marks data stored in a text wave as containing binary rather than text. See **Text Waves Containing Binary Data** on page III-424 for details.

Symbol Font

Igor Pro 6 and before used "system text encoding", typically MacRoman on Macintosh and Windows-1252 on Windows. These text encodings can represent a maximum of 256 characters because each character is represented by a single byte and a single byte can represent only 256 unique values from 0 to 255.

Most Greek letters can not be represented in MacRoman or Windows-1252. Because of that, people resorted to Symbol font when they wanted to display special characters, such as Greek letters. In system text encoding, Symbol font is treated specially. For example, the byte value 0x61 (61 hex, 97 decimal), which is normally interpreted as "small letter a", is interpreted in Symbol font as "Greek small letter alpha".

Thus, in Igor6, to create a textbox that displayed a Greek small letter alpha, you needed to execute this:

```
TextBox "\\F'Symbol'a"
```

Because Igor Pro 7 uses Unicode, you have at your disposal a wide range of characters of all types without changing fonts. To create a textbox that displays a Greek small letter alpha, in Igor7 you simply execute this:

```
TextBox "α"
```

Also, in Igor7, Symbol font is not treated specially. Consequently, in Igor7, this:

```
TextBox "\\F'Symbol'a"
```

displays a "small letter a", not a "small Greek letter alpha". This is a departure from Igor6 and creates compatibility issues.

You can copy Symbol font characters to the clipboard as Unicode using the **Symbol Font Characters** table. You can insert Unicode characters using Edit→Special Characters. In the Add Annotation dialog and in the Axis Label tab of the Modify Axis dialog, you can click Special and choose a character from the Character submenu.

Igor7 Symbol Font Backward Compatibility

The use of Unicode in Igor7 greatly simplifies the display of Greek letters and other special characters. But it creates an incompatibility with Igor6. Without special handling, when Igor7 loads an Igor6 file, Symbol font text meant to display Greek letters would display Roman letters instead. This section explains how Igor7 deals with this issue to provide backward compatibility.

When loading procedure files, including the experiment recreation procedures that execute when you open an Igor experiment file, Igor7 attempts to convert incoming Symbol font characters to Unicode. For compatibility with Igor6, the reverse is done on writing procedure files, including experiment recreation procedures. This conversion is done using a heuristic that involves scanning for certain patterns and consequently is not, and can not be, perfect.

There may be cases where this attempt at backward compatibility creates problems, for example, if you don't care about Igor6 compatibility. If you want to turn Symbol font compatibility off, you can execute

```
SetIgorOption EnableSymbolToUnicode=0
```

and reload the experiment.

If you encounter Symbol font problems and the information below does not provide the solution, please let us know, and provide the original Igor6 text that caused the problem.

Igor7 Symbol Font Backward Compatibility Limitations

An important limitation is that, in the original Igor6 experiment, Symbol font specification must not include anything other than the Symbol characters. No escape sequences, such as font size specifications, are allowed. For example, Igor7 will not handle this Igor6 text:

```
TextBox/C/N=text0/F=0/A=MC "\\Z18A\\F'Symbol'\\Bq\\F]0"
```

because the subscript escape code, "\\B", is inside the Symbol font escape sequence and prevents Igor7 from recognizing this as a Symbol font sequence. As a result, you will see a "q" character instead of the intended small Greek letter theta character.

To enable Igor7 to recognize this as Symbol text, move the subscript escape code outside the Symbol font escape sequence, like this:

```
TextBox/C/N=text0/F=0/A=MC "\\Z18A\\B\\F'Symbol'q\\F]0"
```

When creating graphics in Igor7, for compatibility with Igor6 or for EPS export, use Unicode characters, like this:

```
TextBox/C/N=text0/F=0/A=MC "\\Z18A\\B\\F'Symbol'θ\\F]0"
```

When Igor7 writes this out to a procedure file, it will convert the Symbol font sequence into an Igor6-compatible sequence, by replacing the Unicode theta character with "q".

When creating graphics in Igor7, if you don't care about compatibility with Igor6 or EPS export, you don't need to use Symbol font. You can instead write:

```
TextBox/C/N=text0/F=0/A=MC "\\Z18A\\Bθ"
```

On Macintosh, you might still want to specify Symbol font because it provides almost all of the Symbol characters and you may prefer its style compared to other fonts.

Zapf Dingbat Font

Igor deals with Zapf Dingbats font in the same way. Zapf Dingbats was widely available on Macintosh in previous millennium.

Symbol Tips

On Windows, because Igor7 uses Unicode, Symbol font does not appear to be useful. Consequently, Igor7 substitutes another font when Symbol is encountered.

To insert frequently used symbol characters, choose Edit→Special Characters. This displays Character Viewer on Macintosh and Character Map on Windows, allowing you to insert any Unicode character.

You can copy Symbol font characters to the clipboard as Unicode using the **Symbol Font Characters** table. In the Add Annotation dialog and in the Axis Label tab of the Modify Axis dialog, you can click Special and choose a character from the Character submenu.

Symbol Font Characters

The “Text Encoding.ihf” help file includes a list of Symbol font characters. To display it, execute:

```
DisplayHelpTopic "Symbol Font Characters"
```

You can copy the characters as Unicode from that section of the help file and paste them into another Igor7 window.

Symbols with EPS and Igor PDF

Both EPS and PDF include Symbol font as one of their standard supported fonts. When inserting a character from the above list, you can either specify Symbol font or you can use a Unicode font that supports those characters.

In the case of Symbol font, Igor translates the Unicode code point to the corresponding Symbol single byte code (unless you have specified that even standard fonts be embedded) and such characters in the resulting file will be editable in a program such as Adobe Illustrator. The alternative is to specify a font such as Lucida Sans Unicode in which case the characters are embedded using an outline font and will not be editable.

Be sure to specify either Symbol or a font that will be embedded because it is likely that the current default font is one of the EPS or PDF standard supported fonts. These are not Unicode and the result will not be what you expect.