INSIDE MACINTOSH

Seed Note

Supporting Unicode Input

Seed Note APPLE CONFIDENTIAL
Unreleased Preliminary

October 1, 1998
Technical Publications

Contents

About Unicode Input

Contents

This document describes how applications and input methods can use Text Services Manager 1.5 and Unicode Utilities to support Unicode input on the Mac OS.

IMPORTANT
This is draft documentation. While every effort has been made to ensure accuracy, sections with change bars have not received final technical review.

In addition to Unicode input, most complete Unicode applications will need various other services which will not be covered in this document, such as imaging services for Unicode text as well as processing of this text. While these topics are not addressed here, it is assumed that an application that supports Unicode will use Apple Type Services for Unicode Imaging (ATSUI) to provide the Unicode imaging capability, although Unicode input via the methods described in this document is compatible with any other service for imaging Unicode.

This chapter contains an overview of international text handling on the Mac OS and a more specific introduction to some of the Unicode facilities available with Mac OS 8.5.

A Brief Introduction to International Text on the Mac OS

The following is a quick orientation to some of the terminology used in discussing international text handling in Mac OS 8. If you're already familiar with working with international text on the Mac OS, you can skip this section and go to "About Unicode Text on the Mac OS" (page 11). On the other hand, if you would like more details about Mac OS text handling, you can read *Inside Macintosh:Text*. If you would like more information on Unicode and related topics, see *Inside Macintosh: Programming With the Text Encoding Conversion Manager.*

Languages, Writing Systems, Scripts, and Orthographies

Writing systems and scripts are understood differently in Mac OS 8 and System 7. Mac OS 8 text handling and internationalization software uses the concepts of writing systems and scripts as they are understood in the area of linguistics.

This differs from System 7, in which the concept of a script system and what composed one was particular to System 7. If you have relied on the understanding of these concepts imparted by descriptions of System 7 and its predecessor versions, you'll need to adjust your perspective somewhat to make the transition to international text support in Mac OS 8.

Written representation of a spoken language relies on a writing system. A **writing system**, then, is an artificial construct used to record language in written form. It can be viewed as having three main components—language, scripts, and orthography—with well defined relations to one another.

A **script** comprises a set of symbols that represent the components of a language. A writing system uses one or more scripts for the symbols required to represent linguistic elements, which include sound, meaning, syntax and so forth. A script can be coupled with one language, or it can represent and be used by many languages. Moreover, a language can have more than one script associated with it. For example, the Japanese language uses the Japanese script, while the French, Italian, and Spanish languages all use parts of the Latin script.

A script exists apart from both the languages it represents and the writing systems for which it is used. (A small number of scripts, less than 100, are used by writing systems despite the large number of existing modern and archaic languages.) A special category of scripts, called **pseudoscripts**, exists for use with other scripts. These pseudoscripts include symbols, numbers, and punctuation.

Writing systems can use different scripts at the same time. A writing system uses at least one script and typically one or more pseudoscripts. In this sense, then, it is best to refer to the characters a writing system includes as a **repertoire** of characters, rather than a character set, because these characters can belong to different scripts.

The writing system for a language entails an **orthography** which defines the relationship between the written language and one or more scripts. Among the rules an orthography specifies are rules of directionality, level of discreteness, and units of representation. For example, for mixed-directional text, the direction of a paragraph is important. For writing systems based in European languages, a paragraph is considered a unit of representation, as is a word. Word division and paragraph identification are easily determined for these languages, but this is not necessarily the case for other writing systems, such as those based in Japanese or Indic languages.

Script Systems and Script Codes

Traditionally, on the Mac OS, a **script system** has been understood to be a collection of software facilities that provides for the representation of a specific writing system. This usage of the term “script” in the phrase “script system” should not be confused with the more current, linguistics-derived notion of scripts that is used in Mac OS 8 and described in “Languages, Writing Systems, Scripts, and Orthographies” (page 7).

Types of Mac OS script systems include

- 1-byte simple: small character set, non-contextual, not bidirectional (example: English)
- 1-byte complex: small character set, but with contextual or bidirectional text (example: Devanagari)
- 2-byte: large character set (examples: Japanese, Korean, Chinese, and Simplified Chinese)

At minimum, a script system consists of

- keyboard resources, which provide for text input in any language from any keyboard; these allow for convenient switching from one input language to another on a single keyboard
- international resources, which contain information specific to a particular language, such as its date and time formats, sorting order, and word-break rules
- fonts, that is, sets of glyphs that are associated with specified characters

A **script code** is a numeric value indicating a particular Mac OS script system. Constants are defined for each of the script codes recognized by the Mac OS.

Characters, Character Encodings, and Unicode

A writing system’s alphabet, numbers, punctuation, and other writing marks consist of characters. A **character** is a symbolic representation of an element of a writing system; it is the concept of, for example, “lowercase a” or “number 3”.

In memory, text is stored as **character codes**, where each code is a numeric value that defines a particular character. A **character encoding** is the organization of the set of numeric codes that represent all the meaningful characters of a script system in memory. There are two fundamental classes of Mac OS character encodings: 1-byte and 2-byte.

Unicode is an international standard that combines the characters for all commonly used writing systems into a single, coded character set, based upon a 16-bit character encoding standard. With a universal character encoding such as Unicode, the character sets of separate writing systems do not overlap. Furthermore, Unicode resolves the issue of conflicting character encodings within a single writing system; for example, in Unicode, there is no overlap between Roman character codes and the Symbol font's character codes.

Keyboards and Input Methods

By means of keyboard input, the user can create text that your application stores as character codes.

The system reports the user's key-down, key-up, and auto-key events to your application via the event record. **Key-down** and **key-up** events report that the user pressed or released a key, respectively. **Auto-key** events report that the user has held a key down for a certain amount of time. For keyboard-related events, the application receives both the virtual key code and the character code for the key that is pressed, as well as the state of any **modifier keys** (Shift, Caps Lock, Command, Option, and Control) at the time of the event.

To obtain this information for your application, the Mac OS uses **keyboard resources** to convert keypresses into the correct character codes for the current writing system, taking into account the type of keyboard being used.

Key translation is the process by which character codes are generated. Each keyboard has a particular physical arrangement of keys, and each keypress generates a value called a **raw key code**, which indicates which key was pressed. The keyboard driver that handles the keypress maps these raw key codes to keyboard-independent **virtual key codes**.

Any given script system has one or more keyboard-layout resources. The **keyboard-layout resources** provide script-specific maps for converting a virtual key code into the character code that is passed to your application. As part of the key-translation process, the keyboard-layout resources must take into account the current dead-key state. A **dead key** is a keypress or modifier-plus-keypress combination that produces no immediate character output, but instead affects the character(s) that are ultimately produced by the following keypress(es).

A **keyboard layout** is what the Key Caps application shows. For the purposes of this document, a keyboard-layout resource is the critical item in determining keyboard layout; changing the keyboard layout means changing the keyboard-layout resource. Because keyboard layouts are independent of the physical keyboard attached to the computer, your application has the flexibility of changing text input from one writing system to another by simply using a different keyboard-layout resource.

For languages with large character sets, it is impractical to manufacture keyboards with keys for every possible character. In such a case, it is usually the job of an input method, working in conjunction with a keyboard, to handle text input. An **input method** is a software module, often independent of the application it serves, that performs complex processing of text input, prior to the application's processing of the text. A typical example of an imput method is a translation service that converts character codes that can be entered from the keyboard into character codes that cannot; text input in Japanese, Chinese, and Korean usually requires an input method.

The Text Services Manager

The **Text Services Manager** is the part of the Mac OS that provides an environment for applications to use text services such as input methods. The Text Services Manager handles communication between client applications that request text services and the software modules, known as **text service components**, that provide them. The Text Services Manager presents two separate programming interfaces to the features it provides: one for applications and another for text service components.

While the Mac OS Unicode input architecture allows existing applications to gain the benefits of some forms of Unicode input without undergoing any change whatsoever, full support for Unicode input in a Mac OS application depends on adoption of the text input model provided by the Text Services Manager.

About Unicode Text on the Mac OS

This section provides an overview of the facilities provided by Text Services Manager 1.5 and Unicode Utilities for supporting Unicode text input.

Unicode Script Codes

The set of Mac OS script codes that identify particular script systems now includes Unicode, which is handled as a special Mac OS script code. The Text Encoding Converter and other Mac OS facilities use the constant `kTextEncodingUnicodeDefault` (0x0100) to designate Unicode. However, because some components have only 7 bits available for a script code, rather than the typical 16 bits, the value `smUnicodeScript` (0x7E) can also be used to indicate Unicode. For example, the Text Encoding Converter handles the `smUnicodeScript` value just like `kTextEncodingUnicodeDefault`.

Note
The `smUnicodeScript` symbolic constant is not defined in the 3.2 Universal Interfaces, so your application may have to define this constant itself.

Unicode Keyboard-Layout Resource and the UCKeyTranslate Function

Mac OS 8.5 introduces the **Unicode keyboard-layout resource** (`'uchr'`). Like the (pre-Unicode) keyboard-layout resource (`'KCHR'`), the `'uchr'` resource contains the data necessary to map virtual key codes to character codes for various keyboard layouts. However, the `'uchr'` resource specifies Unicode keyboard layouts—that is, keyboard layouts which produce Unicode character codes, rather than characters in a Mac OS encoding.

Because some Unicode character codes can be mapped to Mac OS encoded character codes (while some cannot), for the purposes of key translation there are considered to be two categories of Unicode keyboard-layout resources. The first category of `'uchr'` resources is one that produces Unicode character codes that are all within the range of a single Mac OS encoding. That is, these **partial Unicode** `'uchr'` resources contain only Unicode characters that can be mapped to characters belonging to the Mac OS encoding associated with its ID range.

The second category of `'uchr'` resources may produce any Unicode characters. That is, these **full Unicode** `'uchr'` resources contain Unicode characters that are either not all within the range of a single Mac OS encoding or are not within the range of any Mac OS encoding. Table 0-1 shows the relationships of keyboard-layout resources to differing types of text input.

Table 0-1 Text input types and keyboard layouts

Input Type	Keyboard Layout
	Resource type, ID
Produces Mac OS encoded characters	KCHR, ≥ 0
Produces partial Unicode characters	uchr, ≥ 0
Produces full Unicode characters	uchr < 0

The function `UCKeyTranslate` is also available as of Mac OS 8.5. In a similar fashion to the (pre-Unicode) `KeyTranslate` function, which used the `'KCHR'` resource to produce character codes, `UCKeyTranslate` uses the `'uchr'` resource to produce Unicode character codes. However, unlike `KeyTranslate`, `UCKeyTranslate` also

1. Outputs multiple character codes. A single keycode (or a dead-key sequence) can produce a string of up to 255 Unicode characters. This facility is useful both for some international script systems and for the production of macros. As an example of the former, the Devanagari keyboard in the Indian Language Kit must be able to produce up to three characters from a single keypress to support the keyboard standards of India.
2. Allows multiple dead keys. The keyboard standards for some countries require double dead keys. For example, Greek keyboards use two dead keys for adding diacritical marks.
3. Handles virtual key codes with a range greater than 0–127. While this requirement is currently uncommon on the Mac OS, some types of keyboards—for example, older Kanji keyboards and keyboards for some other operating systems—may use a larger key code range.
4. Allows virtual key code mapping to depend on keyboard type. While the use of virtual key codes should theoretically remove all dependencies on particular physical keyboards, in some cases key translation does depend on the keyboard type (due to certain scripts, languages, and regions needing subtle differences in layout for specific keyboards). Prior to Mac OS 8.5, the system would use the key-remap resource (`'itlk'`) to map the virtual key codes and modifier state for some key combinations on certain keyboards, before using the `'KCHR'` resource. The `UCKeyTranslate` function accommodates this need by requesting keyboard type information and using the `'uchr'` resource to access the proper keyboard's mapping tables in cases where there

is a keyboard-specific dependency, thus eliminating the need to use the `'itlk'` resource.

Unicode in the Keyboard Menu

The **Keyboard menu** appears on the right side of the menu bar when more than one script system is enabled. It permits the user to choose among keyboard layouts, input methods, and script systems, for text input.

If there are input methods for any of the Mac OS 2-byte script systems that are enabled, the Keyboard menu shows only the input methods; otherwise, in the absence of input methods, it shows the keyboard layouts. For all other enabled script systems, including Unicode, the keyboard menu will show keyboard layouts and input methods.

Note
The Keyboard menu shows each keyboard layout as a single entry, regardless of whether it is specified by a `'KCHR'`, a `'uchr'`, or both.

To display a full Unicode script system in the Keyboard menu, the System file must include an international bundle resource (`'itlb'`) with a resource ID of `smUnicodeScript` (0x7E) and one or more full Unicode keyboard layouts or input methods.

Full Unicode keyboard layouts and input methods (that is, for input sources that produce Unicode characters that are not within the range of a single Mac encoding), if enabled, are shown in their own section of the menu, after all of those for Mac OS script systems.

Supporting Unicode Input in Applications and Input Methods

Contents

This chapter describes how applications and input methods can support Unicode input by using Text Services Manager 1.5 and Unicode Utilities.

- Application developers should read “Supporting Unicode Input in Applications” (page 17) to learn about the steps required for an application to support Unicode input.
- Input method developers should read “Providing Unicode Support in Input Methods” (page 25) to learn about the steps required for an input method to support Unicode input.
- Typically the Text Services Manager calls the `UCKeyTranslate` function when needed. However, there are occasions when an application or input method may need to use this function. See “Using the UCKeyTranslate Function” (page 30) for more details.
- “Creating a 'uchr' Resource” (page 32): This section is under development.

IMPORTANT
This is draft documentation. While every effort has been made to ensure accuracy, sections with change bars have not received final technical review.

Supporting Unicode Input in Applications

In order to support Unicode input, an application must both support the Text Services Manager and request Unicode input.

Applications that do not support Unicode input fall in two categories: those that do not support the Text Services Manager, and those that do, but which do not request Unicode input. In both cases, these applications do receive some of the benefit of text input from new Unicode input sources which can take the form of either Unicode keyboard layouts (specified by `'uchr'` resources) or Unicode input methods and text services.

However, the kinds of Unicode input available to applications that do not support Unicode input are restricted. These applications receive only input from partial Unicode input sources, that is sources that generate only Unicode characters that are all within the repertoire of a single Mac encoding, usually the Mac encoding determined by the current keyboard script. This is because text from partial Unicode input sources is automatically converted by the Text

Services Manager to a Mac OS encoding for delivery to these applications. Full Unicode input sources—that is, those which either generate characters within the repertoire of several Mac encodings or outside the repertoire of any Mac encoding—are not available to these applications and appear disabled in the Keyboard menu.

Supporting the Text Services Manager requires an application to provide Apple event handlers for the full suite of Text Services Manager Apple events in order to support inline input of text. While input can be handled via the bottomline method, this mode of input will not support full Unicode input sources, but only those input sources whose output can be converted to a given Mac encoding (that is, partial Unicode input sources).

Implementing a set of Apple event handlers for the Text Services Manager suite, for the purpose of supporting inline input in general and Unicode input in particular, greatly enhances the text input experience for users of your applications in a variety of existing input sources as well as new Unicode input sources. Even if the majority of existing input methods are associated with a particular Mac script system (and therefore a particular Mac encoding), your application will automatically support these input sources because the Text Services Manager converts all text from Mac OS encoding input sources to Unicode for delivery to applications that have requested Unicode input.

Note
TSMTE does not currently support Unicode input. If an application does rely on TSMTE for input, its input sources will be limited to those which generate input within the repertoire of individual Mac OS encodings.

Identifying an Application as Supporting Unicode

Text Services Manager client applications must create an internal record called a **TSM document** (defined by the `TSMDocument` data type) before they can use any services provided through the Text Services Manager. A TSM document is a private data structure that your application associates with each of its documents that use a text service.

There is a new TSM document type for requesting Unicode input: `kUnicodeDocument` (`'udoc'`). When a Unicode-input TSM document is active, the associated application receives input in Unicode. It can receive input from all input types: full Unicode, partial Unicode, and Mac OS encodings.

Non-Unicode (Mac OS encoded) input is converted to Unicode before being delivered to a Unicode-input TSM document.

When non-Unicode TSM documents are active or when the current application is not a Text Services Manager client, the application receives Mac OS encoded input. In these cases, full Unicode input sources are disabled in the Keyboard menu and cannot be used, and input from partial Unicode sources is automatically converted to the current keyboard script (a Mac OS encoding) by the Text Services Manager.

Your application creates a Unicode TSM document by specifying the `kUnicodeDocument` (`'udoc'`) type in the `supportedInterfaceTypes` parameter of `NewTSMDocument`.

Event Handling for Unicode Text

Text Services Manager 1.5 introduces a new Unicode Apple event that allows applications with Unicode TSM documents to streamline their event handling.

In the old model, your application calls the function `WaitNextEvent` and, when it receives a low-level keyboard event, it passes the event to the Text Services Manager via the `TSMEvent` function. `TSMEvent` then passes the event along to any text service components which might be associated with the document. If a text service component handles the event, `TSMEvent` returns `true`, and your application receives the component-processed character codes in an Apple event. If a text service component does not handle the event, `TSMEvent` returns `false`, and your application handles the event from its `WaitNextEvent` loop.

In the new model, your application still calls `WaitNextEvent` and passes low-level keyboard events to the Text Services Manager via `TSMEvent`. The difference is that `TSMEvent` always returns `true`, to indicate that the key event was processed, either by an input method (and delivered via the standard Text Services Manager Apple events) or by means of direct delivery to the application (via the `kUnicodeNotFromInputMethod` Apple event). Because the `kUnicodeNotFromInputMethod` Apple event contains both the Unicode character code(s) and a copy of the original low-level key event record, your application can now consolidate all of its keyboard input processing in a single logical unit in its Apple event handlers, rather than in its event loop.

This section provides details on how to modify existing Text Services Manager Apple event handlers and discusses the new Text Services Manager Apple event required to support Unicode input. If your application already supports the Text Services Manager, these changes are minimal. If your application does not currently support the Text Services Manager, you should first implement support for the Text Services Manager; see *Inside Macintosh: Text* for details on implementing handlers for Text Services Manager Apple events and user interface guidelines for inline input of text.

Modifying Existing Apple Event Handlers for Unicode

When the active TSM document is of type `kUnicodeDocument`, the Text Services Manager will deliver all text content in Text Services Manager Apple events as Unicode text, in a descriptor whose keyword continues to be `keyAETheData`, but whose descriptor type is `typeUnicodeText`.

When known data structures accompanying the Unicode text contain offsets to text, these offsets are also converted, if needed, to Unicode (byte) offsets to match the encoding of the text delivered to the application's Apple event handler. This delivery of text (and accompanying byte offsets) in Unicode occurs regardless of the type of input source. If the input source is a Unicode input method, text and offsets are passed through by the Text Services Manager to the application's handler unchanged, but if the input source generates text in a Mac encoding, the generated text is converted to Unicode automatically by the Text Services Manager.

Text is converted between Unicode and Mac OS encodings as necessary. Text from Unicode input sources is automatically converted to Mac encodings for delivery to applications that don't use Unicode TSM documents; text from Mac OS encoding input sources is converted to Unicode for delivery to applications using Unicode TSM documents. Similarly, application text requested by an input method (with the Apple event ID `kGetSelectedText`) is converted as necessary.

The Update Active Input Area Event

Your application's Apple event handler for the `kUpdateActiveInputArea` Apple event must obtain the `keyAETheData` parameter using the descriptor type `typeUnicodeText` to obtain the Unicode content of the active input area. The `keyAEFixLength`, `keyAEHiliteRange`, `keyAEUpdateRange`, and `keyAEClauseOffsets` parameters all contain byte offsets into the Unicode text.

For more details on the `kUpdateActiveInputArea` Apple event, see *Inside Macintosh: Text*.

The Position To Offset Event

Your application's Apple event handler for the `kPos2Offset` Apple event must reply with the `keyAEOffset` parameter containing a Unicode text (byte) offset. If the text service requesting the offset is associated with a Mac OS encoding, the Text Service Manager will convert the text offset from Unicode to that of the Mac OS encoding.

For more details on the `kPos2Offset` Apple event, see *Inside Macintosh: Text*.

The Offset To Position Event

Your application's Apple event handler for the `kOffset2Pos` Apple event must treat the `keyAEOffset` parameter as a Unicode text (byte) offset. If the text service specifying the text offset is associated with a Mac OS encoding, the Text Services Manager will convert the text offset from the Mac OS encoding to Unicode before forwarding the Apple event to the application.

For more details on the `kOffset2Pos` Apple event, see *Inside Macintosh: Text*.

The Get Selected Text Event

Your application's Apple event handler for the `kGetSelectedText` Apple event must return the current text selection as Unicode text. If the text service specifying the text offset is associated with a Mac OS encoding, the Text Services Manager will convert the Unicode text to the Mac OS encoding before forwarding the Apple event to the text service. Supporting this event is optional, but recommended.

Note
For a discussion of providing a handler for the Get Selected Text Apple event, see *develop*, Issue 29. This event is not discussed in *Inside Macintosh: Text*.

Supporting the Unicode (Not From Input Method) Apple Event

To support Unicode input via the Text Services Manager, your application must provide a handler for the new Text Services Manager Unicode Apple event whose event ID is `kUnicodeNotFromInputMethod`. When the user generates Unicode input that does not originate from an input method (that is, the Unicode text may be generated by a keyboard layout or is simply not handled by an input method) the Text Services Manager will forward the generated input to your application as Unicode text in the `kUnicodeNotFromInputMethod` Apple event.

Note
Unicode text resulting from input method interactions will be delivered using the `UpdateActiveInputArea` Apple event, as is the case for non-Unicode text.

The `kUnicodeNotFromInputMethod` Apple event contains the Unicode text, a copy of the original low-level key event, and a `ScriptLanguageRecord` structure that identifies the current keyboard script. Your application's event handler for the `kUnicodeNotFromInputMethod` Apple event must obtain the `keyAETheData` parameter using the descriptor type `typeUnicodeText` to obtain the input as Unicode text.

Your application's Apple event handler can also obtain the original low-level key event from a parameter whose keyword is `keyAETSMEventRecord` and whose descriptor type is `typeLowLevelEventRecord`. If the current keyboard layout is determined by a `'KCHR'` resource, you can pass the virtual key code and modifiers to the function `KeyTranslate` to produce a Mac OS encoding character code. Otherwise, if a Unicode keyboard layout is being used (that is, if the keyboard layout is determined by a `'uchr'` resource), you can use the `UCKeyTranslate` function. Typically, you do not need to perform either action.

The application's Apple event handler for the `kUnicodeNotFromInputMethod` event should always fully process the input and return `noErr`. Returning any error or not providing a handler will cause the `TSMEvent` function to indicate that the low-level key event was not handled, in which case your application may not be able to generate the correct text, depending on whether the input source is a Unicode keyboard layout and whether a dead-key sequence is in progress.

Note
Inside Macintosh: Text states that each Text Services Manager Apple event contains two required parameters, one of which is the `keyAEServerInstance` parameter, which identifies the component that is sending the Apple event. In the case of the `kUnicodeNotFromInputMethod` Apple event, this parameter is not included because the event only pertains to cases where a component (such as, an input method) is not handling the data.

Class	`kTextServiceClass`
ID	`kUnicodeNotFromInputMethod`
Requested action	Accept Unicode text.

Required Parameters

Keyword	`keyAETheData`
Descriptor Type	`typeUnicodeText`
Data	Unicode text. Note that this text data has not been processed in any way by a text service component.

Keyword	`keyAETSMEventRecord`
Descriptor Type	`typeLowLevelEventRecord`
Data	A copy of the original low-level key event record.

Keyword	`keyAETSMDocumentRefcon`
Descriptor Type	`typeLongInteger`
Data	A TSM document specifier (reference constant) supplied by the application in a prior call to the `NewTSMDocument` function. This value is associated with the TSM document that is receiving Unicode text input.

Keyword	`keyAETSMScriptTag`
Descriptor Type	`typeIntlWritingCode`
Data	A `ScriptLanguageRecord` structure that identifies the script code and language code associated with the text returned in the `keyAETheData` parameter. If the current input source is partial Unicode, this contains a Mac OS script code. If the current input source is full Unicode, it is 0x7E (`smUnicodeScript`).

Optional Parameters

(none)

Return Parameter

Keyword	`keyErrorNumber`
Descriptor Type:	`typeShortInteger`

Data	Any errors that the application needs to return to the Text Services Manager to terminate processing of the key event that the application passed to `TSMEvent`. `TSMEvent` will return `false` to indicate to the application that the key event was not handled. The application can then attempt to process the event in its event loop. Note that the character code data in the returned key event is not valid in general, but the virtual key code and modifier-key data could still be processed.

Handling Low-Level Keyboard Events for Applications

While low-level keyboard events appear essentially unchanged with Unicode text input, there are certain differences which can affect how text is converted.

Whether or not a Unicode script system is present, the keyboard driver always uses a `'KCHR'` resource to generate the character codes that are posted in the low-level event. Even if the current keyboard layout is specified solely by a `'uchr'` resource, the Script Manager will supply the keyboard driver with the best approximation of an appropriate `'KCHR'` resource to use. However, the resulting character in the low-level event may have no relation to the actual Unicode character, as specified by the `'uchr'` resource. Also, in this case, when the current keyboard layout is specified by a `'uchr'` resource alone, the Text Services Manager disables driver dead-key processing for `'KCHR'` resources and performs all dead-key processing itself.

If the current keyboard layout is specified only by a partial Unicode `'uchr'` resource, and the current application is not using a Unicode TSM document, the Text Services Manager will intercept the key event posted by the driver before it is delivered to the application. The Text Services Manager uses the `'uchr'` resource with the function `UCKeyTranslate` to map the virtual key code and modifiers in the event to a string of Unicode character codes. It will then convert these to character codes in the appropriate Mac OS encoding and post these for delivery to the application in a series of keyboard events. While these appear to your application as normal keyboard events, you cannot automatically reproduce the characters in the events by using the (pre-Unicode) `KeyTranslate` function to convert the key code and modifiers in the event. Instead, you must check to see if a `'uchr'` resource is present to know whether to use `KeyTranslate` or `UCKeyTranslate`.

If the current application is using a Unicode TSM document, the keyboard event posted by the driver is not modified before delivery to the application.

Instead, the application is expected to pass the event to the Text Services Manager via the function `TSMEvent`, which handles all necessary `UCKeyTranslate` calls or conversion to Unicode.

For keyboard layouts that have `'uchr'` resources, `TSMEvent` will use `UCKeyTranslate` to convert the keycode and modifiers in the key event to a sequence of Unicode characters. For keyboard layouts that only have `'KCHR'` resources, `TSMEvent` will convert the Mac OS encoding character in the event to Unicode.

Providing Unicode Support in Input Methods

While existing applications process inline input text in Mac OS encodings, as applications adopt Unicode they will also support input from Unicode input methods, greatly increasing the characters available to the user in individual scripts and offering a convenient and comprehensive environment for multi-script or multilingual text entry. Also, because text contained in Apple events from Unicode input methods does not need to be converted by the Text Services Manager to Unicode for application delivery, the efficiency of inline input processing is greatly improved.

This section identifies the requirements for development of Unicode input methods. While the main requirement imposed by the Text Services Manager is that these input methods communicate externally using Unicode text, the Text Services Manager does not require that an input method perform its internal processing in Unicode nor that the input method image Unicode text in its user interface (input method palettes), although these features are assumed to be desirable or necessary for other reasons.

Text Services Manager 1.5 defines two types of Unicode input methods: full Unicode input methods and partial Unicode input methods. A full Unicode input method is defined to be an input method which may generate Unicode characters outside of the repertoire of any given Mac OS encoding, in multiple Mac OS encoding repertoires, or both. A partial Unicode input method always adheres (externally) to the repertoire of the Mac OS encoding defined by the Mac OS script system to which it belongs.

Partial Unicode input methods appear in the Keyboard menu section for the script to which they belong. Full Unicode input methods and keyboard layouts appear in a new section near the bottom of the Keyboard menu, after the section for Mac OS encodings.

This section describes only the requirements related to Unicode support for input methods. See *Inside Macintosh: Text* for details on implementing classic input methods or for information regarding input methods in general.

Identifying an Input Method as Supporting Unicode

Both partial and full Unicode input methods continue to be Component Manager components, described by the ComponentDescription flags in the component `'thng'` resource. A partial Unicode input method specifies the Mac OS script code with which it is associated, while a full Unicode input method specifies the constant 0x7E (`smUnicodeScript`). Note that while a partial Unicode input method, like a non-Unicode (Mac OS encoding) input method, advertises itself as being associated with a Mac OS script code, it is distinguished by the contents of the `ScriptLanguageRecord` structure that it returns when it responds to a `GetScriptLanguageSupport` call.

While `GetScriptLanguageSupport` has not been a required input method function prior to Text Services Manager 1.5, this is now the mechanism used by the Text Services Manager to distinguish a Mac OS encoding input method from a partial Unicode input method. Since both of these input methods specify a Mac OS script code in the component description flags of the `'thng'` resource, a partial Unicode input method implements its `GetScriptLanguageSupport` function to return an array that includes a `ScriptLanguageRecord` structure with the proper Mac OS language code and a script code of `kTextEncodingUnicodeDefault` (0x0100).

Full Unicode input methods, like non-Unicode input methods, do not need to implement this function, although a full Unicode input method may wish to return an array of `ScriptLanguageRecord` structures, each specifying the `kTextEncodingUnicodeDefault` constant for the script code and the appropriate language code to identify those languages for which it is most suited.

Table 0-2 shows the relationships of keyboard-layout resources and input methods to differing types of text input, including whether the input method must identify the script systems it supports in a `ScriptLanguageRecord` structure to respond to the Text Services Manager function `GetScriptLanguageSupport`.

Table 0-2 Text input types, keyboard layouts, and input method script systems

Input Type	Keyboard Layout	Input Method Script Systems	Input Method Script Systems
	Resource type, ID	As identified by the ComponentDescription flags in the component ('thng') resource	As identified in the ScriptLanguageRecord structure (if necessary)
Produces Mac OS encoded characters	KCHR, ≥ 0	Supply any Mac OS script code (0x00–0x20)	Not necessary, but can supply any Mac OS script code (0x00–0x20)
Produces partial Unicode characters	uchr, ≥ 0	Supply any Mac OS script code (0x00–0x20)	Necessary; must supply the 16-bit Unicode script code (0x0100 = kTextEncodingUnicodeDefault)
Produces full Unicode characters	uchr < 0	Supply the 7-bit Unicode script code (0x7E = smUnicodeScript)	Not necessary, but can supply the 16-bit Unicode script code (0x0100 = kTextEncodingUnicodeDefault)

Responding to the UCTextServiceEvent Function

Partial Unicode and full Unicode input methods are no longer called via the `TextServiceEvent` function. For any Unicode input method, the Text Services Manager always uses the new `UCTextServiceEvent` function.

Like the pre-Unicode `TextServiceEvent` function, `UCTextServiceEvent` specifies the low-level event record, but it also contains the Unicode text stream resulting from the keypress. This is important because the keyboard layout being used may be a Unicode keyboard-layout (`'uchr'`) resource, which may generate more than one character as the result of a single keypress or no characters in the case of a dead-key sequence.

Note that the Text Services Manager forwards the key event to the input method in all cases, even when no output is produced by the `'uchr'` resource. Therefore, the input method should be prepared to be called by the `UCTextServiceEvent` function with just the key event and no Unicode text (`unicodeString=NULL`, `unicodeStrLength=0`). This allows input methods to process Option-Shift equivalents without the need to override the keyboard layout data used by the keyboard driver, as sometimes has been necessary in the past.

Supporting Unicode in Text Services Manager Apple Events

A Unicode input method must transmit all text that is sent via Text Services Manager Apple events as Unicode text, in a descriptor whose keyword continues to be `keyAETheData`, but whose descriptor type is `typeUnicodeText`. All text offsets specified in these Apple events must specify byte offsets into the corresponding Unicode text. This applies to all currently defined Text Services Manager Apple Events: Update Active Input Area, Offset To Position, Position To Offset, and Get Selected Text.

Handling Low-Level Keyboard Events for Input Methods

While low-level keyboard events appear essentially unchanged with Unicode text input, there are certain differences which can affect how text is converted.

Whether or not a Unicode script system is present, the keyboard driver always uses a `'KCHR'` resource to generate the character codes that are posted in the low-level event. Even if the current keyboard layout is specified solely by a `'uchr'` resource, the Script Manager will supply the keyboard driver with the best approximation of an appropriate `'KCHR'` resource to use. However, in the latter case, the resulting character in the low-level event may have no relation to the actual Unicode character as specified by the `'uchr'` resource.

Because keyboard drivers are not equipped to handle a Unicode keyboard-layout (`'uchr'`) resource, which may generate more than one character as the result of a single keypress or no characters in the case of a dead-key sequence, there are three cases where the Text Services Manager disables keyboard driver dead-key processing and performs all dead-key processing itself:

- if an input method of any type is in use
- if the current keyboard layout is specified solely by a `'uchr'` resource (that is, if no `'KCHR'` resource is available)
- if the current document identifies itself as a Unicode TSM document and a `'uchr'` resource is available

In any of these cases, when the Text Services Manager disables dead-key processing in the keyboard driver, it passes each key event to the `UCKeyTranslate` function, whose output is then forwarded to the input method. When a `'uchr'` is not available for input into a Unicode input method, the Text Services Manager relies on the Text Encoding Converter to generate the Unicode characters.

Handling Compatibility Issues

There are two main compatibility issues for Unicode input methods: running on systems with Text Services Manager 1.0 and providing support for applications that do not themselves support Unicode.

Unicode input methods of any kind cannot be selected, and are not loaded, on a system with Text Services Manager 1.0. While this is true of both full Unicode input methods and partial Unicode input methods, a partial Unicode input method could be implemented such that it behaves as a Mac OS encoding input method with Text Services Manager 1.0, and a partial Unicode input method with Text Services Manager 1.5. In the presence of Text Services Manager 1.0, the input method could continue to perform its internal processing in Unicode and convert text to Mac encoding using the Text Encoding Converter either for display in its own palettes (if ATSUI is not available) or for Apple event content. The input method's component description flags specify the Mac script in either world, and, in the presence of Text Services Manager 1.5, the input method may respond to a `GetScriptLanguageSupport` call by returning an array that includes a `ScriptLanguageRecord` structure with the proper Mac OS script code and a language code of `kTextEncodingUnicodeDefault`.

Full Unicode input methods cannot be selected by the user unless the current application's active TSM Document is created with the `kUnicodeDocument` interface type. Until Unicode is adopted to a greater extent, input methods may benefit from restricting Unicode output to the repertoire of a single Mac OS script system, and possibly generate Unicode outside of a Mac encoding's repertoire only when it is certain that the current document is a Unicode TSM document.

Using the UCKeyTranslate Function

In most cases, application and input methods do not need to use `UCKeyTranslate`, because the Text Services Manager automatically calls it when handling input from a Unicode keyboard layout. However, there may be some circumstances when you may wish to call `UCKeyTranslate` directly.

For example, an application may need to determine what character code(s) would have been generated for the virtual key code in the current key-down event if a different modifier-key combination had been used. Listing 0-1 shows how an application could use the function `UCKeyTranslate` to perform its own virtual key code to Unicode character code conversion in an event loop.

Listing 0-1 Using UCKeyTranslate in an event loop

```
#include <MacTypes.h>
#include <Events.h>
#include <LowMem.h>
#include <Resources.h>
#include <Script.h>
#include <UnicodeUtilities.h>

enum {
    kMaxUnicodeInputStringLength = 16
};

main()
    {
    EventRecord *eventPtr;
    Handle uchrHandle;
    UInt32 deadKeyState;
    SInt16 currentKeyScript;
    SInt16 lastKeyLayoutID;
    UniChar unicodeInputString[kMaxUnicodeInputStringLength];
    OSStatus status;
```

```
// initialization
currentKeyScript = GetScriptManagerVariable(smKeyScript);
lastKeyLayoutID = GetScriptVariable(currentKeyScript, smScriptKeys);
deadKeyState = 0;
uchrHandle = GetResource('uchr', lastKeyLayoutID);
…

// event loop
while(true)
    {

    // get next event from WaitNextEvent, then
    switch (eventPtr->what)
        {
        …

        case keyDown:
        case keyUp:
        case autoKey:
            {
            SInt16 currentKeyLayoutID;

            currentKeyScript = GetScriptManagerVariable(smKeyScript);
            currentKeyLayoutID = GetScriptVariable(currentKeyScript, smScriptKeys);
            if (currentKeyLayoutID != lastKeyLayoutID)
                {
                // reset the dead key state if the keyboard layout has changed
                deadKeyState = 0;
                // attempt to get the handle for the new keyboard layout's 'uchr'
                uchrHandle = GetResource('uchr', currentKeyLayoutID);
                lastKeyLayoutID = currentKeyLayoutID;
                }
            // if there is a 'uchr' for the current keyboard layout, use it
            if (uchrHandle != NULL)
                {
                UInt32 keyboardType;
                UInt32 modifierKeyState;
                UInt16 virtualKeyCode;
                UInt16 keyAction;
                UniCharCount actualStringLength;
                virtualKeyCode = ((eventPtr->message) >> 8) & 0xFF;
                keyAction = eventPtr->what - keyDown;
                modifierKeyState = ((eventPtr->modifiers) >> 8) & 0xFF;
                keyboardType = LMGetKbdType();

                status = UCKeyTranslate(*uchrHandle, virtualKeyCode, keyAction,
                                    modifierKeyState, keyboardType, 0,
                                    &deadKeyState, kMaxUnicodeInputStringLength,
                                    &actualStringLength, unicodeInputString);
                // now do something with status and unicodeInputString
                …
                }
            else
                {
                // no 'uchr' resource, do something with 'KCHR'?
                …
                }
            }
            break;
        } // end switch on eventPtr->what
    } // end of while statement for event loop
}
```

Creating a 'uchr' Resource

This section is forthcoming.

Unicode Utilities Reference

Contents

This chapter discusses the Unicode Utilities programming interface in detail.

- "Unicode Utilities Function" (page 35) describes `UCKeyTranslate`.
- "Unicode Utilities Data Types" (page 38) describes the `'uchr'` resource and related structures.
- See also "Unicode Utilities Constants" (page 61) and "Unicode Utilities Result Codes" (page 64).

IMPORTANT
This is draft documentation. While every effort has been made to ensure accuracy, sections with change bars have not received final technical review.

Unicode Utilities Function

UCKeyTranslate

Converts a combination of a virtual key code, a modifier key state, and a dead-key state into a string of one or more Unicode characters.

```
pascal OSStatus UCKeyTranslate (
                    UCKeyboardLayout *keyLayoutPtr,
                    UInt16 virtualKeyCode,
                    UInt16 keyAction,
                    UInt32 modifierKeyState,
                    UInt32 keyboardType,
                    OptionBits keyTranslateOptions,
                    UInt32 *deadKeyState,
                    UniCharCount maxStringLength,
                    UniCharCount *actualStringLength,
                    UniChar unicodeString[] );
```

`keyLayoutPtr`	A pointer to the first element in a resource of type `'uchr'` (page 38). Pass a pointer to the `'uchr'` resource that you wish the `UCKeyTranslate` function to use when converting the virtual key code to a Unicode character. The resource handle associated with this pointer need not be locked, since the `UCKeyTranslate` function does not move memory.
`virtualKeyCode`	An unsigned 16-bit integer. Pass a value specifying the virtual key code that is to be translated. For ADB keyboards, virtual key codes are in the range from 0 to 127.
`keyAction`	An unsigned 16-bit integer. Pass a value specifying the current key action. See “Key Action Constants” (page 63) for descriptions of possible values.
`modifierKeyState`	An unsigned 32-bit integer. Pass a bit mask indicating the current state of various modifier keys. You may obtain this value from the `modifiers` field of the event record as follows: `modifierKeyState = ((EventRecord.modifiers) >> 8) & 0xFF;`
`keyboardType`	An unsigned 32-bit integer. Pass a value specifying the physical keyboard type (that is, the keyboard shape shown by Key Caps). You may call the low-memory accessor function `LMGetKbdType` for this value.
`keyTranslateOptions`	A bit mask of options for controlling `UCKeyTranslate`. See “Key Translation Options Constants” (page 64) for descriptions of possible values.
`deadKeyState`	A pointer to an unsigned 32-bit value, initialized to zero. `UCKeyTranslate` uses this value to store private information about the current dead key state.
`maxStringLength`	A value of type `UniCharCount`. Pass the number of 16-bit Unicode characters that are contained in the buffer passed in the `unicodeString` parameter. This may be a value of up to 255, although it would be rare to get more than 4 characters.

`actualStringLength`
: A pointer to a value of type `UniCharCount`. On return this value contains the actual number of Unicode characters placed into the buffer passed in the `unicodeString` parameter.

`unicodeString[]`
: An array of `UniChar` values. Pass a pointer to the buffer whose sized is specified in the `maxStringLength` parameter. On return, the buffer contains a string of Unicode characters resulting from the virtual key code being handled. The number of characters in this string is less than or equal to the value specified in the `maxStringLength` parameter.

function result A result code. See "Unicode Utilities Result Codes" (page 64). If you pass `NULL` in the `keyLayoutPtr` parameter, `UCKeyTranslate` returns `paramErr`. `UCKeyTranslate` also returns `paramErr` for an invalid `'uchr'` resource format or for invalid `virtualKeyCode` or `keyAction` values, as well as for `NULL` pointers to output values. The result `kUCOutputBufferTooSmall` (-25340) is returned for an output string length greater than `maxStringLength`.

DISCUSSION

The `UCKeyTranslate` function uses the data in a Unicode keyboard-layout (`'uchr'`) resource to map a combination of virtual key code and modifier key state to a sequence of up to 255 Unicode characters. This mapping process depends on, and may update, a dead key state; the `UCKeyTranslate` function and the `'uchr'` resource support multiple dead keys. The mapping may also depend on the specific type of key action and the type of physical keyboard being used. The `UCKeyTranslate` function supports non-ADB keyboards, an extensible set of modifier keys, and other possible extensions.

In most cases, your application does not need to call `UCKeyTranslate`, since the Text Services Manager will automatically call it on your behalf to handle input from a Unicode keyboard layout. However, there may be some circumstances in which your application should call `UCKeyTranslate`. For example, your application may need to determine what character(s) would have been generated for the virtual key code in the current key-down event if a different modifier-and-key combination had been used.

Unicode Utilities Data Types

'uchr'

The Unicode keyboard-layout (`'uchr'`) resource contains the data necessary to map virtual key codes to Unicode character codes for a given keyboard layout. Each installed script system has one or more keyboard-layout resources, which may be of type `'uchr'` or `'KCHR'` (an older, non-Unicode keyboard-layout format). There may be one or more keyboard-layout resources for each language or region, depending upon the preference of the user.

The `'uchr'` resource ID is determined as for the `'KCHR'` resource, with one exception. That is, typically, the resource ID for each Unicode keyboard-layout resource is within the range of resource ID numbers for its script system. The ID number of the default keyboard-layout resource for a script system is specified in the `itlbKeys` field of the script's international bundle (`'itlb'`) resource. The exception to this is that if a given `'uchr'` resource specifies any Unicode characters that are not within the range of a single Mac OS encoding (or are not in any Mac OS encoding), then you must use a negative number for the resource.

For a given resource ID, the system may contain a `'KCHR'` resource, a `'uchr'` resource, or both. If both a `'KCHR'` resource and a `'uchr'` resource are present, they must have the same ID, and the `'uchr'` resource should match the behavior of the `'KCHR'` resource. The keyboard menu shows each keyboard layout as a single entry, regardless of whether it is specified by a `'KCHR'`, a `'uchr'`, or both.

Note
The `'uchr'` resource contains offsets to tables that may be in any order. Because of the complexity of this format, Rez may not readily be used to create `'uchr'` resources. A `'uchr'` resource may be created in any data-description language that allows the specification of arbitrary binary data.

The UCKeyTranslate Function and the Unicode Keyboard-Layout Resource

You pass a pointer to a `'uchr'` resource to the function `UCKeyTranslate` (page 35) so that the function may use the resource to translate a virtual key code and produce a string of up to 255 Unicode characters on return.

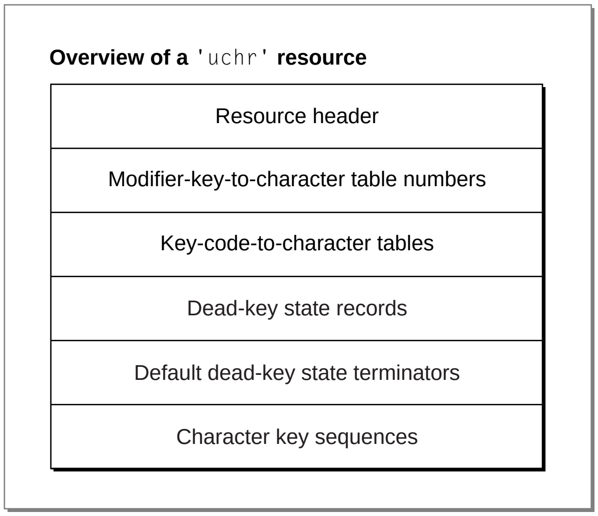
The basic process by which `UCKeyTranslate` uses the `'uchr'` resource to translate virtual key codes into Unicode characters is as follows. For details on specific steps, see the descriptions of the various sections of the `'uchr'` resource in "The Unicode Keyboard-Layout Resource Format" (page 39) and the descriptions for each of the specific types used in the resource that are referenced below.

5. The bit pattern specifying the modifier key state is mapped by the `UCKeyModifiersToTableNum` (page 56) structure to a table number.
6. The table number maps to an offset within a `UCKeyToCharTableIndex` (page 57) structure that refers to the actual key-code-to-character tables.
7. The key-code-to-character tables map the virtual key code to `UCKeyOutput` (page 47) values, for which there are two possibilities:
   - If bits 15 and 14 of the `UCKeyOutput` value are 01, the `UCKeyOutput` value is an index into the offsets contained in a `UCKeyStateRecordsIndex` (page 58) structure. If this occurs, the mapping process for the virtual key code continues on to Step 4.
   - Otherwise, the `UCKeyOutput` value produces one or more Unicode characters, either directly or via reference to a `UCKeySequenceDataIndex` (page 60) structure. This ends the mapping process for a given virtual key code.
8. The offsets in a `UCKeyStateRecordsIndex` structure refer to `UCKeyStateRecord` (page 49) dead-key state records.
9. The dead-key state records map from the current dead-key state to one or more Unicode characters to be output or the following dead-key state (if any). The mapping process for a given virtual key code may end with the dead-key state record or, if there is no dead-key state record entry for the key code, with a default state terminator, as specified in the resource's `UCKeyStateTerminators` (page 59) table.

The Unicode Keyboard-Layout Resource Format

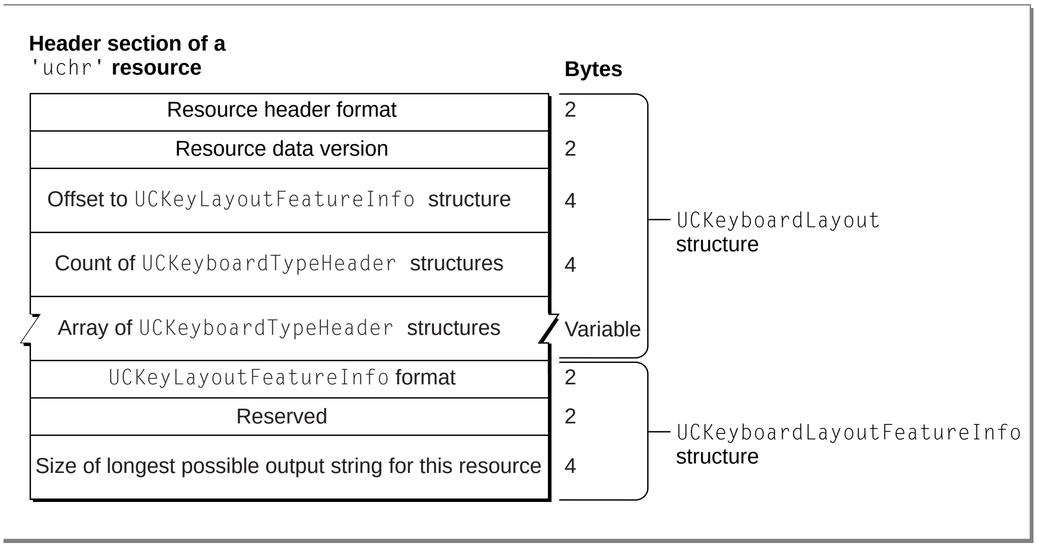
The `'uchr'` format consists of a header information section and five key mapping data sections, as shown in Figure 0-1.

Figure 0-1 'uchr' resource layout


Overview of a 'uchr' resource
Resource header
Modifier-key-to-character table numbers
Key-code-to-character tables
Dead-key state records
Default dead-key state terminators
Character key sequences

The header section of a compiled `'uchr'` resource contains a structure of type `UCKeyboardLayout` (page 53) and an optional structure of type `UCKeyLayoutFeatureInfo` (page 56). See Figure 0-2 for an illustration of this section.

Figure 0-2 'uchr' resource header


Header section of a 'uchr' resource
Bytes
Resource header format
2
Resource data version
2
Offset to UCKeyLayoutFeatureInfo structure
4
Count of UCKeyboardTypeHeader structures
4
Array of UCKeyboardTypeHeader structures
Variable
UCKeyLayoutFeatureInfo format
2
Reserved
2
Size of longest possible output string for this resource
4
UCKeyboardLayout structure
UCKeyboardLayoutFeatureInfo structure

The elements in the header section of a `'uchr'` resource are

- the resource header format
- the version of the data in this resource
- an offset to a `UCKeyLayoutFeatureInfo` structure, if any
- a count of the `UCKeyboardTypeHeader` structures that follow
- an array of structures of type `UCKeyboardTypeHeader` (page 54); each `UCKeyboardTypeHeader` entry specifies a range of physical keyboard types and contains offsets to each of the key mapping sections to be used for that range of keyboard types
  - first keyboard type in this entry
  - last keyboard type in this entry
  - offset to the `UCKeyModifiersToTableNum` structure (required)
  - offset to the `UCKeyToCharTableIndex` structure (required)
  - offset to the `UCKeyStateRecordsIndex` structure (optional, may be 0 if there is no table)

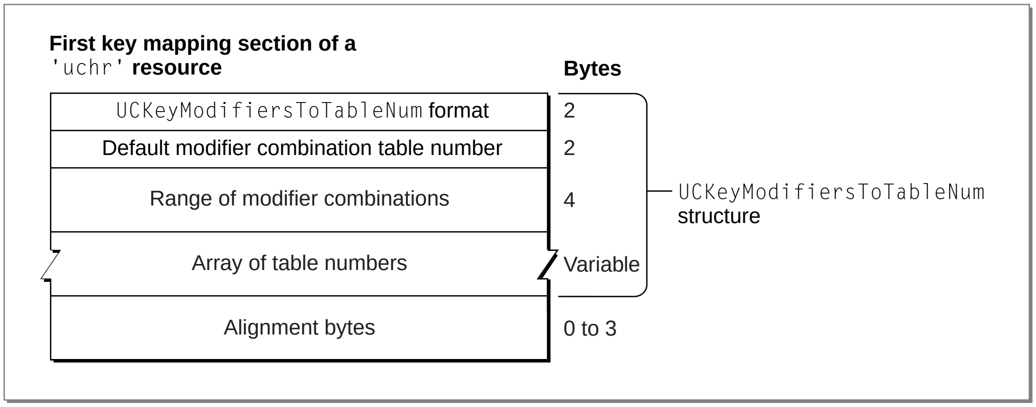
- □ offset to the `UCKeyStateTerminators` structure (optional, may be 0 if there is no table)
- □ offset to the `UCKeySequenceDataIndex` structure (optional, may be 0 if there is no table)

- ■ the format of the `UCKeyLayoutFeatureInfo` structure
- ■ a reserved field
- ■ a value of type `UniCharCount`, specifying the longest possible output string to be produced by this `'uchr'` resource

There may be a variable number of each of the following `'uchr'` key mapping sections.

The first key mapping section contains a structure of type `UCKeyModifiersToTableNum` (page 56), which maps a modifier key combination to a particular key-code-to-character table number; and alignment bytes. There may be multiple instances of this entire key mapping section. See Figure 0-3 for an illustration of this section.

Figure 0-3 'uchr' modifier combination to key-code-to-character table number map


First key mapping section of a 'uchr' resource
Bytes
UCKeyModifiersToTableNum format
2
Default modifier combination table number
2
Range of modifier combinations
4
Array of table numbers
Variable
Alignment bytes
0 to 3
UCKeyModifiersToTableNum structure

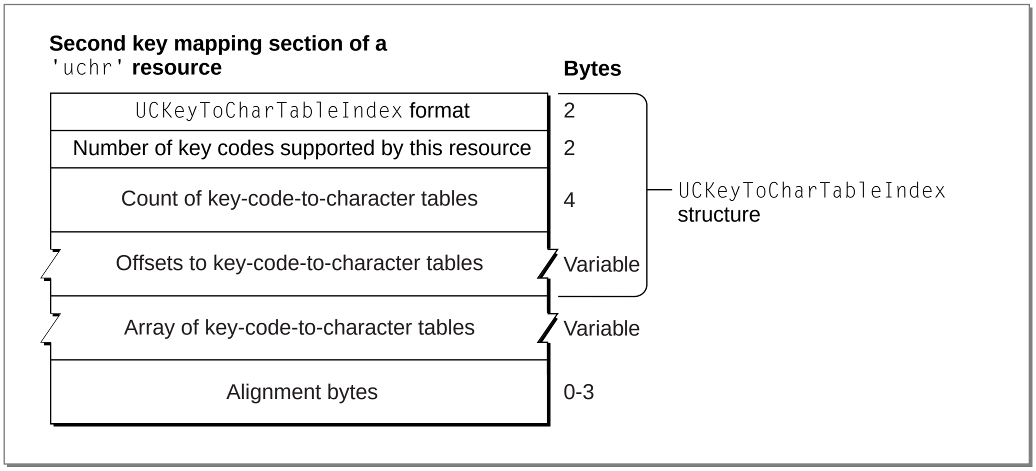
The elements in the first key mapping section of a `'uchr'` resource are

- ■ the format of the `UCKeyModifiersToTableNum` structure

- the table number for modifier combinations that are outside of the range of the `tableNum` field's array; that is, the default (fallback) table number
- the range of modifier bit combinations for which there are entries in the `tableNum` field's array
- an array of indexes into the key-code-to-character table offsets contained in the `UCKeyToCharTableIndex` structure in the next section
- alignment bytes (to a 4-byte boundary)

The second key mapping section contains a structure of type `UCKeyToCharTableIndex` (page 57); the list of key-code-to-character tables, each of which maps a virtual key code to a 16-bit `UCKeyOutput` value; and alignment bytes. There may be multiple instances of this entire key mapping section. See Figure 0-4 for an illustration of this section.

Figure 0-4 'uchr' key-code-to-character tables


Second key mapping section of a 'uchr' resource
Bytes
UCKeyToCharTableIndex format
2
Number of key codes supported by this resource
2
Count of key-code-to-character tables
4
Offsets to key-code-to-character tables
Variable
Array of key-code-to-character tables
Variable
Alignment bytes
0-3
UCKeyToCharTableIndex structure

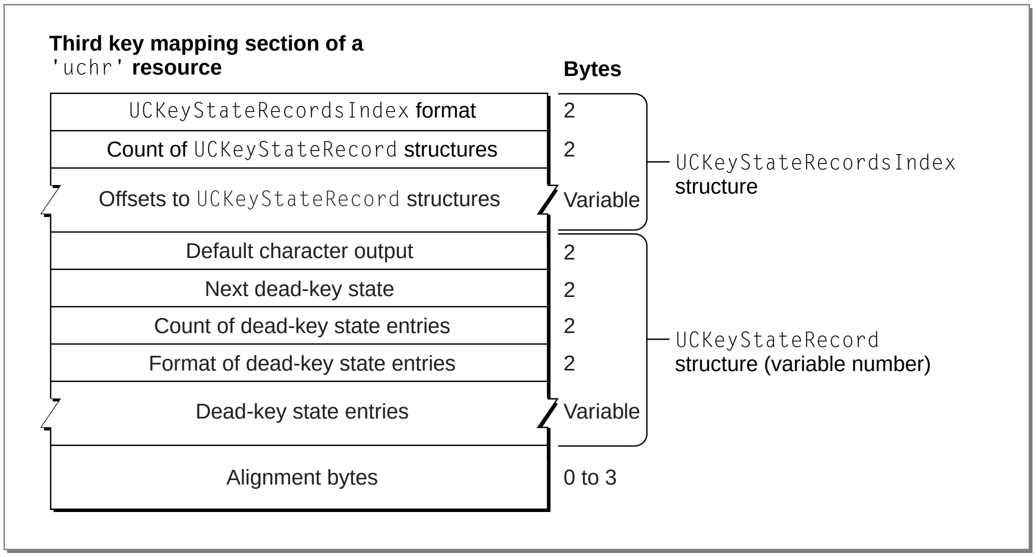
The elements in the second key mapping section of a `'uchr'` resource are

- the format of the `UCKeyToCharTableIndex` structure
- the number of virtual key codes supported by this resource
- a count of the key-code-to-character tables

- an array of offsets from the beginning of the resource to each of the key-code-to-character tables
- an array of key-code-to-character tables containing values of type `UCKeyOutput` (page 47)
- alignment bytes (to a 4-byte boundary)

The third key mapping section is a map to dead-key state records. It contains a structure of type `UCKeyStateRecordsIndex` (page 58), which is an index to `UCKeyStateRecord` structures; a variable number of dead-key state records of type `UCKeyStateRecord` (page 49); and alignment bytes. There may be multiple instances of this entire key mapping section (or 0; this section need not be present if no `UCKeyOutput` value refers to a dead-key state record). See Figure 0-5 for an illustration of this section.

Figure 0-5 'uchr' dead-key state records


Third key mapping section of a 'uchr' resource
Bytes
UCKeyStateRecordsIndex format
2
Count of UCKeyStateRecord structures
2
Offsets to UCKeyStateRecord structures
Variable
UCKeyStateRecordsIndex structure
Default character output
2
Next dead-key state
2
Count of dead-key state entries
2
Format of dead-key state entries
2
Dead-key state entries
Variable
UCKeyStateRecord structure (variable number)
Alignment bytes
0 to 3

The elements in the third key mapping section of a `'uchr'` resource are

- the format of the `UCKeyStateRecordsIndex` structure

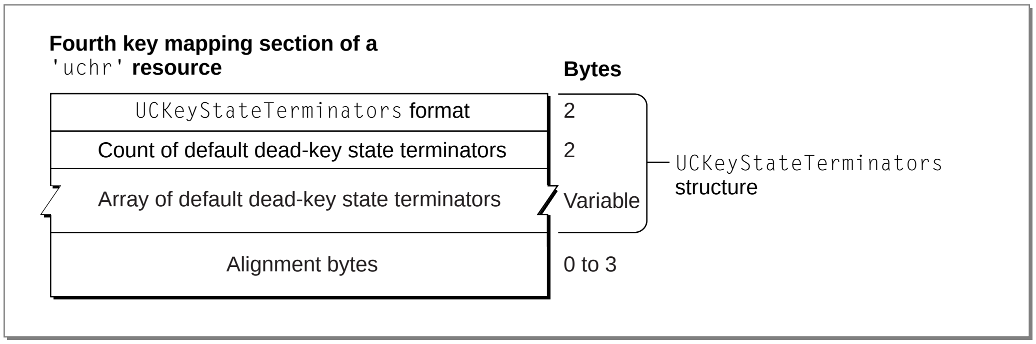
- a count of the dead-key state records to follow
- an array of offsets from the beginning of the resource to each of the `UCKeyStateRecord` values following

Immediately following the `UCKeyStateRecordsIndex` structure are a variable number of values of type `UCKeyStateRecord` (page 49). Any keycode-modifier combination which initiates a dead-key state or which is a valid terminator of a dead-key state refers to one of these records. However, these records also permit more complex dead-key state processing, such as a series of transitions from one dead-key state directly into another in which each transition can emit a sequence of one or more Unicode characters. Each record contains

- a value of type `UCKeyCharSeq` (page 48) specifying the character(s) produced by the input keycode when no dead-key state is currently in effect
- a value specifying the dead-key state produced by the input keycode when no dead-key state is currently in effect
- a count of the elements in the `stateEntryData[]` field's array
- the format of the data in the `stateEntryData[]` field's array
- an array of dead-key state entry data; each entry maps from the current dead-key state to the character(s) that are produced or to the following dead-key state, if any
- alignment bytes (to a 4-byte boundary)

The fourth key mapping section contains a structure of type `UCKeyStateTerminators` (page 59) and alignment bytes. This is an optional list of default terminators (characters or sequences) for each state; if this table is not present or does not extend far enough to have a terminator for the state, nothing is output when the state terminates. There may be multiple (or 0) instances of this entire key mapping section. See Figure 0-6 for an illustration of this section.

Figure 0-6 'uchr' default dead-key state terminators

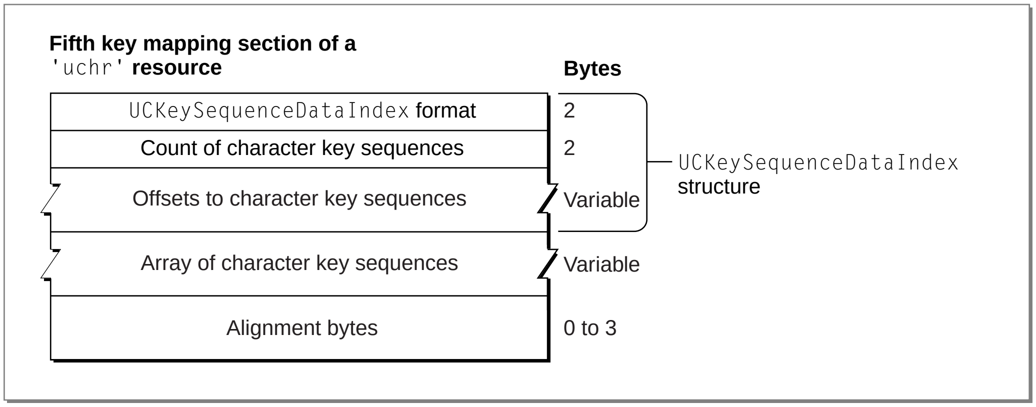

Fourth key mapping section of a 'uchr' resource
Bytes
UCKeyStateTerminators format
2
Count of default dead-key state terminators
2
Array of default dead-key state terminators
Variable
Alignment bytes
0 to 3
UCKeyStateTerminators structure

The elements in the fourth key mapping section of a `'uchr'` resource are

- the format of the `UCKeyStateTerminators` structure
- a count of default dead-key state terminators contained in the `keyStateTerminators` field's array
- an array of default dead-key state terminators, described as values of type `UCKeyCharSeq` (page 48)
- alignment bytes (to a 4-byte boundary)

The fifth key mapping section of the resource is an optional list of character sequences; it contains a structure of type `UCKeySequenceDataIndex` (page 60) and Unicode character sequences. This permits a single keypress to generate a sequence of characters, or to generate a single character outside the range that can be represented directly by a `UCKeyOutput` (page 47) or `UCKeyCharSeq` (page 48) value. There may be multiple (or 0) instances of this entire key mapping section. See Figure 0-7 for an illustration of this section.

Figure 0-7 'uchr' character key sequences


Fifth key mapping section of a 'uchr' resource
Bytes
UCKeySequenceDataIndex format
2
Count of character key sequences
2
Offsets to character key sequences
Variable
Array of character key sequences
Variable
Alignment bytes
0 to 3
UCKeySequenceDataIndex structure

The elements in the fifth key mapping section of a `'uchr'` resource are

- the format of the `UCKeySequenceDataIndex` structure
- a count of the Unicode character sequences that follow the `UCKeySequenceDataIndex` structure
- an array of offsets from the beginning of the `UCKeySequenceDataIndex` structure to the Unicode character sequences that follow it
- an array of Unicode character sequences
- alignment bytes (to a 4-byte boundary)

UCKeyOutput

The `UCKeyOutput` type is a 16-bit value used in the second key mapping section of the `'uchr'` (page 38) resource to specify values in key-code-to-character tables. You use a `UCKeyOutput` value in a key-code-to-character table to represent one of the following:

- an index to a dead-key state record
- an index to a Unicode character sequence
- a single Unicode character

```
typedef UInt16 UCKeyOutput;
```

The interpretation of a `UCKeyOutput` value depends on bits 15 and 14.

If they are 01 (that is, for values in the range of 0x4000–0x7FFF), then bits 0–13 are an index into the `keyStateRecordOffsets[]` field of a `UCKeyStateRecordsIndex` (page 58) structure, which contains offsets to a separate resource-wide list of dead-key state records.

If they are 10 (that is, for values in the range of 0x8000–0xBFFF), then bits 0–13 are an index into the `charSequenceOffsets[]` field of a `UCKeySequenceDataIndex` (page 60) structure, which contains offsets to a separate resource-wide list of Unicode character sequences. If a `UCKeySequenceDataIndex` structure is not present in the resource or the index is beyond the end of the list, then the entire value (that is, bits 0–15) is a single Unicode character to emit.

Otherwise (for values in the range of 0x0000–0x3FFF and 0xC000–0xFFFD), bits 0–15 are a single Unicode character, with the exception that a value of 0xFFFE–0xFFFF means no character output (these are invalid Unicodes).

Most single Unicode characters that are likely to be generated by direct keyboard input are in the range 0x0000–0x33FF or 0xE000–0xFFFD, and so are covered by the single-character cases above. Characters outside this range can still be generated by direct keyboard input—in which case they must be represented as 1-character sequences. The fifth key mapping section of the `'uchr'` resource, introduced by the `UCKeySequenceDataIndex` (page 60) type, provides for this option.

UCKeyCharSeq

The `UCKeyCharSeq` type is a 16-bit value used in the third key mapping section of the `'uchr'` (page 38) resource to specify the output of a dead-key state. Specifically, the dead-key state record—a structure of type `UCKeyStateRecord` (page 49)—uses a `UCKeyCharSeq` value to contain the character output that results from the resolution of a given dead-key state. You can use a `UCKeyCharSeq` value in a dead-key state record to represent one of the following:

- an index to a Unicode character sequence
- a single Unicode character

`UCKeyCharSeq` is similar to `UCKeyOutput` (page 47), but does not itself support indices into dead-key state records.

```
typedef UInt16 UCKeyCharSeq;
```

The interpretation of `UCKeyCharSeq` depends on bits 15 and 14.

If they are 10 (that is, for values in the range of 0x8000–0xBFFF), then bits 0–13 are an index into the `charSequenceOffsets[]` field of a `UCKeySequenceDataIndex` (page 60) structure, which contains offsets to a separate resource-wide list of Unicode character sequences. If a `UCKeySequenceDataIndex` structure is not present in the resource or the index is beyond the end of the list, then the entire value (that is, bits 0–15) is a single Unicode character to emit.

Otherwise (for values in the range of 0x0000–0x7FFF and 0xC000–0xFFFD), bits 0–15 are a single Unicode character, with the exception that a value of 0xFFFE–0xFFFF means no character output (these are invalid Unicodes).

UCKeyStateRecord

The following is the `UCKeyStateRecord` type, which is used in the third key mapping section of the `'uchr'` (page 38) resource to determine dead-key state transitions. The `UCKeyStateRecord` structure permits complex dead-key state processing, such as a series of transitions from one dead-key state directly into another, in which each transition can emit a sequence of one or more Unicode characters.

Any modifier key combination which initiates a dead-key state or which is a valid terminator of a dead-key state refers to one of these records via the `UCKeyOutput` (page 47) values in key-code-to-character tables. A `UCKeyOutput` value may index the offsets contained in a `UCKeyStateRecordsIndex` (page 58) structure, which in turn refers to the actual dead-key state records.

Each `UCKeyStateRecord` structure maps from the current dead-key state to the character data to be output or the following dead-key state (if any), as follows:

- If the current dead-key state is zero (that is, there are no dead keys in effect) the value in `stateZeroCharData` is output and the state is set to the value in `stateZeroNextState` (this can be used to initiate a dead-key state).

- If the current dead-key state is non-zero and there is an entry for the state in `stateEntryData`, then the corresponding value in `stateEntryData.charData` is output. The next state is then set to either a `kUCKeyStateEntryTerminalFormat` or a `kUCKeyStateEntryRangeFormat` value; in either case, if the next dead-key state is 0, this implements a valid dead-key state terminator.
- If the current dead-key state is non-zero, and there is no entry for the state in `stateEntryData`, the default state terminator is output from the `'uchr'` resource's `UCKeyStateTerminators` (page 59) table for the current state (or nothing may be output, if there is no `UCKeyStateTerminators` table or it has no entry for the current state). Then the value in `stateZeroCharData` is output, and the state is set to the value in `stateZeroNextState`.

```
struct UCKeyStateRecord {
    UCKeyCharSeq    stateZeroCharData;
    UInt16          stateZeroNextState;
    UInt16          stateEntryCount;
    UInt16          stateEntryFormat;
    UInt32          stateEntryData[kVariableLengthArray];
};
```

Field descriptions

`stateZeroCharData`	A value of type `UCKeyCharSeq` (page 48) specifying the Unicode character(s) produced from a given key code while no dead-key state is in effect.
`stateZeroNextState`	An unsigned 16-bit integer specifying the dead-key state produced from a given key code when no previous dead-key state is in effect. If the `UCKeyStateRecord` structure does not intiate a dead-key state (but only provides terminators for other dead-key states), this will be 0. A non-zero value specifies the resulting new dead-key state and refers to the current state entry within the `stateEntryData[]` field for the following dead-key state record that is applied.
`stateEntryCount`	An unsigned 16-bit integer specifying the number of elements in the `stateEntryData[]` field's array for a given dead-key state record.
`stateEntryFormat`	An unsigned 16-bit integer specifying the format of the data in the `stateEntryData[]` field's array. This should be 0 if the `stateEntryCount` field is set to 0. Currently available

	values are `kUCKeyStateEntryTerminalFormat` and `kUCKeyStateEntryRangeFormat`; see “Key State Entry Format Constants” (page 62) for descriptions of these values.
`stateEntryData[]`	An array of dead-key state entries, whose size depends on their format, but which will always be a multiple of 4 bytes. Each entry maps from the current dead-key state to the Unicode character(s) that result when a given character key is pressed or to the next dead-key state, if any. The format of the entry is specified by the `stateEntryFormat` field to be either that of type `UCKeyStateEntryTerminal` (page 51) or `UCKeyStateEntryRange` (page 52).

UCKeyStateEntryTerminal

The following is the `UCKeyStateEntryTerminal` type, which is used in the `stateEntryData[]` field of the `UCKeyStateRecord` (page 49) structure. You should use the `UCKeyStateEntryTerminal` format for simple dead-key states that are terminated by a single keystroke, as in the U.S. keyboard layout. Each entry maps from the current dead-key state to the Unicode character(s) produced when a given character key is pressed that terminates the dead-key state.

```
struct UCKeyStateEntryTerminal {
    UInt16          curState;
    UCKeyCharSeq    charData;
};
```

Field descriptions

`curState`	An unsigned 16-bit integer specifying the current dead-key state.
`charData`	A value of type `UCKeyCharSeq` (page 48) specifying the Unicode character(s) produced when a given character key is pressed.

UCKeyStateEntryRange

The following is the `UCKeyStateEntryRange` type, which is used in the `stateEntryData[]` field of the `UCKeyStateRecord` (page 49) structure. You should use the `UCKeyStateEntryRange` format for complex (multiple) dead-key states.

For each virtual key code, an entry in its dead-key state record maps from the current dead-key state to the Unicode character(s) produced or to the next dead-key state, as follows.

If the current dead-key state is within a valid dead-key state range for the given input character—that is, if its value is greater than or equal to `curStateStart` and less than or equal to `curStateStart + curStateRange`—then

- If the base `charData` value for the given dead-key state range is in the range of valid Unicode characters, a character is produced and the dead-key state may be terminated.

and/or

- If the base `nextState` value is not 0, a new dead-key state is produced.

In the first case, the output character is determined as follows: The base `charData` value is incremented by the resulting product of (the difference between the current state and the start of that state's range) and (a multiplier). That is: `charData += (curState - curStateStart) * deltaMultiplier`

Similarly, in the second case, the resulting dead-key state, which is the new `curState` value, is determined as follows: The base `nextState` value is incremented by the resulting product of (the difference between the current state and the start of that state's range) and (a multiplier). That is: `nextState += (curState - curStateStart) * deltaMultiplier`

The fields of the `UCKeyStateEntryRange` structure are briefly described below.

```
struct UCKeyStateEntryRange {
    UInt16          curStateStart;
    UInt8           curStateRange;
    UInt8           deltaMultiplier;
    UCKeyCharSeq    charData;
    UInt16          nextState;
};
```

Field descriptions

`curStateStart`	An unsigned 16-bit integer specifying the beginning of a given dead-key state range.
`curStateRange`	An unsigned 8-bit integer specifying the number of entries in a given dead-key state range.
`deltaMultiplier`	An unsigned 8-bit integer.
`charData`	A value of type `UCKeyCharSeq` (page 48). This base character value is used to determine the actual Unicode character(s) produced when a given dead-key state terminates.
`nextState`	An unsigned 16-bit integer. This base dead-key state value is used to determine the following dead-key state, if any.

UCKeyboardLayout

The following is the `UCKeyboardLayout` type, which is used in the `'uchr'` (page 38) resource header. It specifies version and format information, offsets to the various subtables, and an array of `UCKeyboardTypeHeader` entries.

```
struct UCKeyboardLayout {
    UInt16                  keyLayoutHeaderFormat;
    UInt16                  keyLayoutDataVersion;
    ByteOffset              keyLayoutFeatureInfoOffset;
    ItemCount               keyboardTypeCount;
    UCKeyboardTypeHeader    keyboardTypeList[kVariableLengthArray];
};
```

Field descriptions

`keyLayoutHeaderFormat`	An unsigned 16-bit integer identifying the format of the structure. Set to `kUCLayoutHeaderFormat`.
`keyLayoutDataVersion`	An unsigned 16-bit integer identifying the version of the data in the resource, in binary code decimal format. For example, 0x0100 would equal version 1.0.
`keyLayoutFeatureInfoOffset`	An unsigned 32-bit integer providing an offset to a structure of type `UCKeyLayoutFeatureInfo` (page 56), if such

	is used in the resource. May be 0 if no `UCKeyLayoutFeatureInfo` table is included in the resource.
`keyboardTypeCount`	An unsigned 32-bit integer specifying the number of `UCKeyboardTypeHeader` structures in the `keyboardTypeList[]` field's array.
`keyboardTypeList[]`	A variable-length array containing structures of type `UCKeyboardTypeHeader` (page 54). Each `UCKeyboardTypeHeader` entry specifies a range of physical keyboard types and contains offsets to each of the key mapping sections to be used for that range of keyboard types.

UCKeyboardTypeHeader

The following is the `UCKeyboardTypeHeader` type, which specifies a range of physical keyboard types and contains offsets to each of the key mapping sections to be used for that range of keyboard types. Typically, you use an array of `UCKeyboardTypeHeader` structures, of which the first entry in the array is the default and will be used if the keyboard type does not fall within the range for any other entry. See `UCKeyboardLayout` (page 53) for a further discussion of the context for use of the `UCKeyboardTypeHeader` type.

```
struct UCKeyboardTypeHeader {
    UInt32      keyboardTypeFirst;
    UInt32      keyboardTypeLast;
    // The next 5 fields are offsets to the five key mapping sections
    ByteOffset  keyModifiersToTableNumOffset;
    ByteOffset  keyToCharTableIndexOffset;
    ByteOffset  keyStateRecordsIndexOffset;
    ByteOffset  keyStateTerminatorsOffset;
    ByteOffset  keySequenceDataIndexOffset;
};
```

Field descriptions

`keyboardTypeFirst`	An unsigned 32-bit integer specifying the first keyboard type in this entry. For the initial entry (that is, the default entry) in an array of `UCKeyboardTypeHeader` structures, you should set this value to 0. The initial `UCKeyboardTypeHeader`

entry will be used if the keyboard type passed to `UCKeyTranslate` does not match any other entry, that is, if it is not within the range of values specified by `keyboardTypeFirst` and `keyboardTypeLast` for any entry.

`keyboardTypeLast` An unsigned 32-bit integer specifying the last keyboard type in this entry. For the initial entry (that is, the default entry) in an array of `UCKeyboardTypeHeader` structures, you should set this value to 0.

`keyModifiersToTableNumOffset`
An unsigned 32-bit integer providing an offset to a structure of type `UCKeyModifiersToTableNum` (page 56). The `'uchr'` resource requires a `UCKeyModifiersToTableNum` structure, therefore this field must contain a non-zero value.

`keyToCharTableIndexOffset`
An unsigned 32-bit integer providing an offset to a structure of type `UCKeyToCharTableIndex` (page 57). The `'uchr'` resource requires a `UCKeyToCharTableIndex` structure, therefore this field must contain a non-zero value.

`keyStateRecordsIndexOffset`
An unsigned 32-bit integer providing an offset to a structure of type `UCKeyStateRecordsIndex` (page 58), if such is used in the resource. This value may be 0 if no dead-key state records are included in the resource.

`keyStateTerminatorsOffset`
An unsigned 32-bit integer providing an offset to a structure of type `UCKeyStateTerminators` (page 59), if such is used in the resource. This value may be 0 if no dead-key state terminators are included in the resource.

`keySequenceDataIndexOffset`
An unsigned 32-bit integer providing an offset to a structure of type `UCKeySequenceDataIndex` (page 60), if such is used in the resource. This value may be 0 if no character key sequences are included in the resource.

UCKeyLayoutFeatureInfo

The following is the `UCKeyLayoutFeatureInfo` type, which is used in the header section of the `'uchr'` (page 38) resource.

```
struct UCKeyLayoutFeatureInfo {
    UInt16          keyLayoutFeatureInfoFormat;
    UInt16          reserved;
    UniCharCount    maxOutputStringLength;
};
```

Field descriptions

`keyLayoutFeatureInfoFormat`	An unsigned 16-bit integer identifying the format of the `UCKeyLayoutFeatureInfo` structure. Set to `kUCKeyLayoutFeatureInfoFormat`.
`reserved`	Reserved. Set to 0.
`maxOutputStringLength`	An unsigned 32-bit integer specifying the longest possible output string of Unicode characters to be produced by this `'uchr'` resource.

UCKeyModifiersToTableNum

The following is the `UCKeyModifiersToTableNum` type, which is used in the first key mapping section of the `'uchr'` (page 38) resource. It maps a modifier key combination to a particular key-code-to-character table number.

```
struct UCKeyModifiersToTableNum {
    UInt16      keyModifiersToTableNumFormat;
    UInt16      defaultTableNum;
    ItemCount   modifiersCount;
    UInt8       tableNum[kVariableLengthArray];
};
```

Field descriptions

keyModifiersToTableNumFormat	An unsigned 16-bit integer identifying the format of the UCKeyModifiersToTableNum structure. Set to kUCKeyModifiersToTableNumFormat.
defaultTableNum	An unsigned 16-bit integer identifying the table number to use for modifier combinations that are outside of the range included in the tableNum[] field.
modifiersCount	An unsigned 32-bit integer specifying the range of modifier bit combinations for which there are entries in the tableNum[] field.
tableNum[]	An array of unsigned 8-bit integers that map modifier bit combinations to table numbers. These values are indexes into the keyToCharTableOffsets[] array in UCKeyToCharTableIndex (page 57); these, in turn, are offsets to the actual key-code-to character tables, which follow the UCKeyToCharTableIndex structure in the resource.

UCKeyToCharTableIndex

The following is the UCKeyToCharTableIndex type, which is used in the second key mapping section of the 'uchr' (page 38) resource. It precedes the list of key-code-to-character tables, each of which maps a key code to a 16-bit UCKeyOutput (page 47) value.

```
struct UCKeyToCharTableIndex {
    UInt16      keyToCharTableIndexFormat;
    UInt16      keyToCharTableSize;
    ItemCount   keyToCharTableCount;
    ByteOffset  keyToCharTableOffsets[keyToCharTableCount];
};
```

Field descriptions

keyToCharTableIndexFormat	An unsigned 16-bit integer identifying the format of the UCKeyToCharTableIndex structure. Set to kUCKeyToCharTableIndexFormat.

`keyToCharTableSize`
: An unsigned 16-bit integer specifying the number of virtual key codes supported by this resource; for ADB keyboards this is 128 (with virtual key codes ranging from 0 to 127).

`keyToCharTableCount`
: An unsigned 32-bit integer specifying the number of key-code-to-character tables, typically 6 to 12.

`keyToCharTableOffsets[]`
: An array of offsets from the beginning of the `'uchr'` resource to each of the `UCKeyOutput` (page 47) key-code-to-character tables in the `keyToCharData[]` array that follows this structure in the resource.

UCKeyStateRecordsIndex

The following is the `UCKeyStateRecordsIndex` type, which is used in the third key mapping section of the `'uchr'` (page 38) resource. The `UCKeyStateRecordsIndex` structure is an index to dead-key state records of type `UCKeyStateRecord` (page 49). Any keycode-modifier combination which initiates a dead-key state or which is a valid terminator of a dead-key state refers to one of these records, via the `UCKeyStateRecordsIndex` structure.

```
struct UCKeyStateRecordsIndex {
    UInt16      keyStateRecordsIndexFormat;
    UInt16      keyStateRecordCount;
    ByteOffset  keyStateRecordOffsets[keyStateRecordCount];
};
```

Field descriptions

`keyStateRecordsIndexFormat`
: An unsigned 16-bit integer identifying the format of the `UCKeyStateRecordsIndex` structure. Set to `kUCKeyStateRecordsIndexFormat`.

`keyStateRecordCount`
: An unsigned 16-bit integer specifying the number of dead-key state records that are included in the resource.

`keyStateRecordOffsets[]`

An array of offsets from the beginning of the resource to each of the `UCKeyStateRecord` (page 49) values that follow this structure in the `'uchr'` resource.

UCKeyStateTerminators

The following is the `UCKeyStateTerminators` type, which is used in the fourth key mapping section of the `'uchr'` (page 38) resource. The `UCKeyStateTerminators` structure contains the list of default terminators (characters or sequences) for each dead-key state that is handled by a `'uchr'` (page 38) resource. When a dead-key state is in effect but a modifier-and-key combination is typed which has no special handling for that state, the default terminator for the state is output before the modifier-and-key combination is processed. If this table is not present or does not extend far enough to have a terminator for the state, nothing is output when the state terminates.

```
struct UCKeyStateTerminators {
    UInt16          keyStateTerminatorsFormat;
    UInt16          keyStateTerminatorCount;
    UCKeyCharSeq    keyStateTerminators[keyStateTerminatorCount];
};
```

Field descriptions

`keyStateTerminatorsFormat`

An unsigned 16-bit integer identifying the format of the `UCKeyStateTerminators` structure. Set to `kUCKeyStateTerminatorsFormat`.

`keyStateTerminatorCount`

An unsigned 16-bit integer specifying the number of default dead-key state terminators contained in the `keyStateTerminators[]` array.

`keyStateTerminators[]`

An array of default dead-key state terminators, described as values of type `UCKeyCharSeq` (page 48); `keyStateTerminators[0]` is the terminator for state 1, and so on.

UCKeySequenceDataIndex

The following is the `UCKeySequenceDataIndex` type, which is used in the fifth key mapping section of the `'uchr'` (page 38) resource. The `UCKeySequenceDataIndex` structure contains offsets to a list of character sequences for the `'uchr'` resource. This permits a single keypress to generate a sequence of characters, or to generate a single character outside the range that can be represented directly by a `UCKeyOutput` or `UCKeyCharSeq` value.

```
struct UCKeySequenceDataIndex {
    UInt16  keySequenceDataIndexFormat;
    UInt16  charSequenceCount;
    UInt16  charSequenceOffsets[charSequenceCount+1];
};
```

Field descriptions

`keySequenceDataIndexFormat`
An unsigned 16-bit integer identifying the format of the `UCKeySequenceDataIndex` structure. Set to `kUCKeySequenceDataIndexFormat`.

`charSequenceCount` An unsigned 16-bit integer specifying the number of Unicode character sequences that follow the end of the `UCKeySequenceDataIndex` structure.

`charSequenceOffsets[]`
An array of offsets from the beginning of the `UCKeySequenceDataIndex` structure to the Unicode character sequences that follow it. Because a given offset indicates both the beginning of a new character sequence and the end of the sequence that precedes it, the length of each sequence is determined by the difference between the offset to that sequence and the value of the next offset in the array. The array contains one more entry than the number of character sequences; the final entry is the offset to the end of the final character sequence.

Unicode Utilities Constants

Key Output Index Masks

You can use the following masks to test the bits in values of type `UCKeyOutput` (page 47).

```
enum {
    kUCKeyOutputStateIndexMask       = 0x4000,
    kUCKeyOutputSequenceIndexMask    = 0x8000,
    kUCKeyOutputTestForIndexMask     = 0xC000,    // test bits 14-15
    kUCKeyOutputGetIndexMask         = 0x3FFF     // get bits 0-13
};
```

Constant descriptions

`kUCKeyOutputStateIndexMask`
: If the bit specified by this mask is set, the `UCKeyOutput` value contains an index into a `UCKeyStateRecordsIndex` structure.

`kUCKeyOutputSequenceIndexMask`
: If the bit specified by this mask is set, the `UCKeyOutput` value contains an index into a `UCKeySequenceDataIndex` structure.

`kUCKeyOutputTestForIndexMask`
: You can use this mask to test the bits in the `UCKeyOutput` value that determine whether the value contains an index to any other structure. If both bits specified by this mask are clear, the `UCKeyOutput` value does not contain an index to any other structure.

`kUCKeyOutputGetIndexMask`
: You can use this mask to test the bits in a `UCKeyOutput` value that provide the actual index to another structure.

Key State Entry Format Constants

The following constants are used in structures of type `UCKeyStateRecord` (page 49) to indicate the format for dead-key state records.

```
enum {
    kUCKeyStateEntryTerminalFormat  = 0x0001,
    kUCKeyStateEntryRangeFormat     = 0x0002
};
```

Constant descriptions

`kUCKeyStateEntryTerminalFormat`
: Specifies that the entry format is that of the `UCKeyStateEntryTerminal` (page 51) structure. Use this format for simple (single) dead-key states, as in the U.S. keyboard layout.

`kUCKeyStateEntryRangeFormat`
: Specifies that the entry format is that of the `UCKeyStateEntryRange` (page 52) structure. Use this format for complex (multiple) dead-key states, as in the hex input and Hangul input keyboard layouts.

Key Format Code Constants

The following constants are those currently defined to be used within the various structures in a `'uchr'` (page 38) resource to indicate each structure’s format.

```
enum {
    kUCKeyLayoutHeaderFormat        = 0x1002,
    kUCKeyLayoutFeatureInfoFormat   = 0x2001,
    kUCKeyModifiersToTableNumFormat = 0x3001,
    kUCKeyToCharTableIndexFormat    = 0x4001,
    kUCKeyStateRecordsIndexFormat   = 0x5001,
    kUCKeyStateTerminatorsFormat    = 0x6001,
    kUCKeySequenceDataIndexFormat   = 0x7001
};
```

Constant descriptions

`kUCKeyLayoutHeaderFormat`	The format of a structure of type `UCKeyboardLayout` (page 53).
`kUCKeyLayoutFeatureInfoFormat`	The format of a structure of type `UCKeyLayoutFeatureInfo` (page 56).
`kUCKeyModifiersToTableNumFormat`	The format of a structure of type `UCKeyModifiersToTableNum` (page 56).
`kUCKeyToCharTableIndexFormat`	The format of a structure of type `UCKeyToCharTableIndex` (page 57).
`kUCKeyStateRecordsIndexFormat`	The format of a structure of type `UCKeyStateRecordsIndex` (page 58).
`kUCKeyStateTerminatorsFormat`	The format of a structure of type `UCKeyStateTerminators` (page 59).
`kUCKeySequenceDataIndexFormat`	The format of a structure of type `UCKeySequenceDataIndex` (page 60).

Key Action Constants

You can supply the following constants for the `keyAction` parameter of the function `UCKeyTranslate` (page 35) to indicate the current key action.

```
enum {
    kUCKeyActionDown    = 0,
    kUCKeyActionUp      = 1,
    kUCKeyActionAutoKey = 2,
    kUCKeyActionDisplay = 3
};
```

Constant descriptions

`kUCKeyActionDown`	The user is pressing the key.
`kUCKeyActionUp`	The user is releasing the key.

`kUCKeyActionAutoKey`
: The user has the key in an "auto-key" pressed state; that is, the user is holding down the key for an extended period of time and is thereby generating multiple key strokes from the single key.

`kUCKeyActionDisplay`
: The user is requesting information for key display, as in the Key Caps application.

Key Translation Options Constants

The following are the currently defined bit assignments and masks for the `keyTranslateOptions` parameter of the function `UCKeyTranslate` (page 35).

```
enum {
    kUCKeyTranslateNoDeadKeysBit    = 0
    kUCKeyTranslateNoDeadKeysMask   = 1L << kUCKeyTranslateNoDeadKeysBit
};
```

Constant descriptions

`kUCKeyTranslateNoDeadKeysBit`
: The bit number of the bit that turns off dead-key processing. This prevents setting any new dead-key states, but allows completion of any dead-key states currently in effect.

`kUCKeyTranslateNoDeadKeysMask`
: The mask for the bit that turns off dead-key processing. This prevents setting any new dead-key states, but allows completion of any dead-key states currently in effect.

Unicode Utilities Result Codes

The Unicode Utilities result codes are listed below.

`kUCOutputBufferTooSmall`	-25340	Output buffer too small for Unicode string result

What's New With Text Services Manager 1.5

Contents

What’s New With Text Services Manager 1.5

This appendix covers what’s new in Text Services Manager 1.5 For information about Text Services Manager 1.0, see *Inside Macintosh: Text*.

IMPORTANT
This appendix is an incomplete preliminary draft. While every effort has been made to ensure accuracy, sections with change bars have not received final technical review.

About Text Services Manager 1.5

The Text Services Manager was originally released with System 7.1. Version 1.0 of the Text Services Manager provided support for input methods in a limited set of script systems (Japanese, Korean, and Simplified and Traditional Chinese). Text Services Manager 1.5 supports input methods for all script systems available on the Mac OS, as well as Unicode, and is fully PowerPC native.

Text Services Manager 1.5 Reference

For descriptions of Text Services Manager 1.5 `Gestalt` selectors, see “Gestalt Selectors for Text Services Manager 1.5” (page A-68).

For descriptions of Text Services Manager 1.5 functions, see “Functions for Text Services Manager 1.5” (page A-68).

For descriptions of Text Services Manager 1.5 constants, see “Constants for Text Services Manager 1.5” (page A-69).

Gestalt Selectors for Text Services Manager 1.5

The following gestalt selector and value can be used to determine the version of the Text Services Manager.

```
enum {
    gestaltTSMgrVersion = 'tsmv',
    gestaltTSMgr15      = 0x0150
};
```

Discussion to come.

Functions for Text Services Manager 1.5

UCTextServiceEvent

Routes an event to a specified Unicode-supportive text service component.

```
pascal ComponentResult UCTextServiceEvent (
                     ComponentInstance ts,
                     short numOfEvents,
                     EventRecord *event,
                     UniChar unicodeString[],
                     UniCharCount unicodeStrLength);
```

`ts`	A value of type `ComponentInstance` identifying the currently executing instance of the text service component.
`numOfEvents`	A short integer value specifying the number of events that the Text Services Manager is passing.
`event`	A pointer to the event record for the event that the Text Services Manager is passing to the text service.
`unicodeString[]`	An array of `UniChar` values. The Text Services Manager passes the text service the string of Unicode characters resulting from

the virtual key code being handled. The number of characters in this string is equal to the value specified in the `unicodeStrLength` parameter.

`unicodeStrLength`
A value of type `UniCharCount`. The number of 16-bit Unicode characters contained in the buffer passed in the `unicodeString[]` parameter.

function result A value of type `ComponentResult`. If the text service component handles the event, it should return a nonzero value and change the event to a null event. If it does not handle the event, it should return 0.

DISCUSSION

Partial Unicode and full Unicode input methods are no longer called via the `TextServiceEvent` function. For any Unicode input method, the Text Services Manager always uses the new `UCTextServiceEvent` function.

The Text Services Manager forwards the key event to the input method in all cases, even when no output is produced by the `'uchr'` resource. Therefore, the input method should be prepared to be called by the `UCTextServiceEvent` function with just the key event and no Unicode text (`unicodeString=NULL`, `unicodeStrLength=0`). This allows input methods to process Option-Shift equivalents without the need to override the keyboard layout data used by the keyboard driver, as sometimes has been necessary in the past.

NewCServiceWindow

To come.

Constants for Text Services Manager 1.5

Unicode Document and Text Service Constants

Discussion to come.

```
enum {
    kUnicodeDocument    = 'udoc',
        /* TSM Document type for Unicode-savvy application */
    kUnicodeTextService = 'utsv'
        /* Component type for Unicode Text Service */
};
```

Language and Script Constants

Discussion to come.

```
enum {
    kUnknownLanguage= 0xFFFF,
    kUnknownScript  = 0xFFFF,
    kNeutralScript  = 0xFFFF
};
```

Unicode Text Services Manager Apple Event Constants

Discussion to come.

```
kUnicodeNotFromInputMethod'unim'; Unicode text when event not handled by
Input Method or no Input Method
```

```
; AppleScript 1.3: New Text types
typeUnicodeText'utxt'
typeStyledUnicodeText'sutx'
```

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Line art was created using Adobe™ Illustrator and Adobe Photoshop.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Adobe Letter Gothic.

WRITER
Donna S. Lee

Acknowledgments to Jim Chan, Andy Daniels, Peter Edberg, Michael Grady, and Eric Simenel.