Technical Note TN2056

Installable Keyboard Layouts

CONTENTS

Mac OS X 10.2 adds the ability to install a new keyboard layout by putting a file or bundle in a standard folder.

It is also now possible to define a Unicode keyboard via an XML text file. These keyboards have the same set of capabilities as the `'uchr'` resource keyboards defined in the document "Supporting Unicode Input." In fact, the XML file is translated to the uchr format and then handled in exactly the same way.

[Sep 27 2002]

Installation

Keyboard layouts can be installed in one of the following locations:

```
/Library/Keyboard Layouts/

~/Library/Keyboard Layouts/

/Network/Library/Keyboard Layouts/
```

Resources in `/Library/` are shared among all users on a given machine. Resources in `~/Library/` are visible only to that user. Resources in `/Network/Library/` are shared among all users on a network.

Keyboards have ID numbers which uniquely identify them; these are not shown to the end user. If the system detects two or more keyboards with the same ID number, it will renumber keyboards as necessary to eliminate the conflict.

After installing a new keyboard, the user must log out and then log in again to make the keyboard available.

Back to top

Supported Keyboard Data Formats

There are several different formats that are supported:

1. Old-style Keyboard Suitcase (resource file)

   Keyboard.rsrc (resource file containing `KCHR`, `uchr`, `itlk`, `kcns`, `kcs#`, `kcs4`, `kcs8` resources)

   The name of the keyboard comes from the resource name of the `KCHR`/`uchr`, and cannot be localized. The encoding used is that of the script to which the keyboard belongs, or UTF-8 for Unicode keyboards (negative IDs). The resource file may have one or more keyboards in it. The resources may be in either the resource fork or the data fork (not both); the file extension must be `.rsrc`.

   Icons may be in the kcns resource format, which is new for Mac OS X and is the same as the icns icon format, or in

the older `kcs#/kcs4/kcs8` format used on Mac OS 9. If no icon resources are present, the keyboard will have a generic icon in the input menu and in International Preferences.

To use a Mac OS 9 keyboard suitcase, append the file extension `.rsrc`.

2. Localized Keyboard Suitcase Bundle

```
Roman.bundle/
    Contents/
        Info.plist
        Resources/
            MyLayouts.rsrc
            English.lproj/
                InfoPlist.strings
        version.plist
```

```
InfoPlist.strings file contents:
    "U.S." = "U.S.";
    "Australian" = "Australian";
    "Austrian" = "Austrian";
    ...
```

The `Info.plist` and `version.plist` files follow standard bundle conventions. The bundle identifier must take the form `com.apple.keyboardlayout.`*name*.

The `MyLayouts.rsrc` file (which may have any name but must have the extension `.rsrc`) contains sets of `KCHR`, `uchr`, `itlk`, `kcns`, `kcs#`, `kcs4`, and `kcs8` resources.

The resource file may have one or more keyboards in it; icons are as for format 1. The keyboard layout names may be localized via the `InfoPlist.strings` file within the various `.lproj` folders. The unlocalized key (left hand side) will be the same as the keyboard resource name, in Unicode rather than its native encoding. For example, a Roman `KCHR` resource would have its resource name in MacRoman, but the key in the `InfoPlist.strings` file must be the Unicode equivalent.

This format supports resources in the data fork only.

3. XML Keyboard Definition

This must be a valid text XML file following the specification later in this document, and must have the extension `.keylayout`. As discussed below, the name is always in Unicode, but it cannot be localized.

`Mykeyboard.keylayout`

After installation, an XML keyboard will become available in the Input Menu pane of International Preferences if no errors were encountered in compiling it. If there is an error in the file, an error message will be written to `console.log`, prefixed by "uchr XML compiler." If the error is an XML syntax error, the line in the file where the error occurred is given. Usually, only the first error encountered is diagnosed, and parsing is aborted.

Since `console.log` is erased when you log out and log in, to see error messages, force the keyboard to compile by opening International Preferences and switching to the Input Menu pane. Launching an application and typing will also force all keyboards to compile if the modification date of the directory has changed. If the keyboard does not appear in International Preferences, you can then check in `console.log` for error messages. To fource the modification date on the directory to change, you can use the Unix `touch` command.

Keyboards in this format can have an optional associated icon file, in the standard `icns` format; it has the same name as the keyboard file with the extension `.icns`:

`Mykeyboard.icns`

If this file is absent, the keyboard will have a generic icon in the input menu and in International Preferences.

4. Localized XML Keyboard Bundle

```
        MyKeyboards.bundle/
            Contents/
                Info.plist
                Resources/
                    MyKeyboard1.keylayout
                    MyKeyboard1.icns
                    MyKeyboard2.keylayout
                    MyKeyboard2.icns
                    English.lproj/
                        InfoPlist.strings
                    version.plist
```

```
        InfoPlist.strings file contents:
            "MyKeyboard1" = "MyKeyboard1";
            "MyKeyboard2" = "MyKeyboard2";
```

This format supports localized names for XML keyboards. The bundle may contain multiple XML files ending in `.keylayout`. Localization is as for format 2 above. The `.icns` file is optional for each keyboard; if it is absent, the keyboard has a generic icon in the input menu and in International Preferences.

The rest of this document discusses the new XML keyboard data format.

Back to top

XML DTD

XML keyboards must follow the XML DTD (Document Type Definition) below:

```
<!-- Overall structure -->
<!ELEMENT keyboard (layouts+, modifierMap+, keyMapSet+, actions*, terminators*)>
<!ATTLIST keyboard group NMTOKEN #REQUIRED >
<!ATTLIST keyboard id NMTOKEN #REQUIRED >
<!ATTLIST keyboard name CDATA #REQUIRED >
<!ATTLIST keyboard maxout NMTOKEN #IMPLIED >

<!-- Hardware layout elements -->
<!ELEMENT layouts (layout+) >
<!ELEMENT layout EMPTY >
<!ATTLIST layout first NMTOKEN #REQUIRED >
<!ATTLIST layout last NMTOKEN #REQUIRED >
<!ATTLIST layout modifiers IDREF #REQUIRED >
<!ATTLIST layout mapSet IDREF #REQUIRED >

<!-- Modifier descriptions -->
<!ELEMENT modifierMap (keyMapSelect+) >
<!ATTLIST modifierMap id ID #REQUIRED >
<!ATTLIST modifierMap defaultIndex NMTOKEN #REQUIRED >

<!ELEMENT keyMapSelect (modifier+) >
<!ATTLIST keyMapSelect mapIndex NMTOKEN #REQUIRED >

<!ELEMENT modifier EMPTY >
<!ATTLIST modifier keys CDATA #REQUIRED >

<!-- Keyboard mapping -->
<!ELEMENT keyMapSet (keyMap+) >
<!ATTLIST keyMapSet id ID #REQUIRED >

<!ELEMENT keyMap (key+) >
<!ATTLIST keyMap index NMTOKEN #REQUIRED >
<!ATTLIST keyMap baseMapSet IDREF #IMPLIED >
<!ATTLIST keyMap baseIndex NMTOKEN #IMPLIED >

<!ELEMENT key (action*) >
<!ATTLIST key code NMTOKEN #REQUIRED >
<!ATTLIST key output CDATA #IMPLIED >
<!ATTLIST key action IDREF #IMPLIED >

<!-- Actions (state records) -->
<!ELEMENT actions (action+) >
<!ELEMENT action (when+) >
<!ATTLIST action id ID #IMPLIED >

<!ELEMENT when EMPTY >
<!ATTLIST when state NMTOKEN #REQUIRED >
<!ATTLIST when through NMTOKEN #IMPLIED >
<!ATTLIST when output CDATA #IMPLIED >
<!ATTLIST when multiplier NMTOKEN #IMPLIED >
<!ATTLIST when next NMTOKEN #IMPLIED >

<!-- Terminators -->
<!ELEMENT terminators (when+) >
```

XML File Structure

A keyboard layout description is a standard XML file, and so follows the XML specification. The text file encoding should be Unicode, either UTF-8 or UTF-16. It's a good idea to have both `xml` and `DOCTYPE` directives at the beginning of the file, to clearly identify it:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE keyboard SYSTEM "file://localhost/System/Library/DTDs/KeyboardLayout.dtd">
```

\<keyboard\> Element

The top level element is `<keyboard>`. `<keyboard>` has the following attributes, all of which are required:

group   The section of the keyboard menu in which this layout should appear. Right now, this is the script code. Unicode keyboards have a script code of 126.

id   The unique ID for the keyboard. Right now, this is numeric, and must match the script bundle specified in "`group`". Unicode keyboards have negative IDs. If this ID collides with that of another keyboard, the system will assign a new ID. See further discussion below.

name   The name of the keyboard, to appear in the Keyboard Menu. This is in whatever encoding the keyboard file itself is in (e.g., UTF-8 or UTF-16), not the encoding corresponding to the script of the keyboard. Note that this name cannot be localized, but it is possible to localize the names of keyboards using the bundle mechanism.

The following attribute is optional:

maxout   The maximum number of UTF-16 values that can be generated from one keypress.

The keyboard element must contain exactly one `<layouts>` element, one or more `<modifierMap>` elements, one or more `<keyMapSet>` elements, an optional `<actions>` element, and an optional `<terminators>` element.

Unicode Keyboards (uchr or XML) can be divided into two classes. The first are those associated with a particular Mac OS script code. These keyboards should normally only generate Unicode strings that can be converted to the encoding associated with the script. The scripts (and associated codes) supported in Mac OS X are Roman (0), Japanese (1), Simplified Chinese (25), Traditional Chinese(2), Korean (3), Cyrillic (7), and Central European (29); there are no plans to support any other scripts. This type of keyboard is available to both Unicode and non-Unicode (WorldScript) applications.

It is possible for a keyboard associated with a script to generate Unicode strings that cannot be converted to the associated encoding. This has the disadvantage that such strings will cause question marks (?) to appear in non-Unicode applications. For example, the Kotoeri Japanese input method can generate any Unicode character, but characters outside MacJapanese are only supported in Unicode applications.

The second class of keyboards generate Unicode characters not associated with any of the scripts listed above. Such keyboards are only available to Unicode applications, and have negative id values. They should be assigned to group 126.

A good rule of thumb in deciding whether to associate a keyboard with a script code or not is whether "typical" text typed with the keyboard fits entirely within the script's character encoding. It is best to be conservative as users will have trouble understanding why they are getting ?'s in their WorldScript applications.

\<layouts\> Element

The `<layouts>` element has no attributes, and contains one or more `<layout>` elements. Collectively, these elements define which `<modifierMap>` and `<keyMapSet>` elements control the mapping of keys for particular hardware keyboards.

The `<layout>` element which is physically first controls the mapping of keyboards whose IDs are not contained within the range of IDs for any `<layout>` element.

Apple will map new keyboard hardware IDs to one of the existing ones, so it is usually sufficient to copy and paste the entire `<layouts>` element from an existing keyboard layout.

\<layout\> Element

The `<layout>` element is an empty element (i.e., it has no subelements and is written in the empty form: `<layout />`). It defines which `<modifierMap>` and `<keyMapSet>` to use for a particular range of hardware keyboard IDs. It has the following attributes, all of which are required:

first   The hardware ID of the first keyboard type controlled by this element.

last   The hardware ID of the last keyboard type controlled by this element

modifiers   The identifier of the `<modifierMap>` element to use for this range of hardware keyboard types.

mapSet   The identifier of the `<mapSet>` element to use for this range of hardware keyboard types.

Example:

```
<layouts>
 <layout first="0" last="17" modifiers="commonModifiers" mapSet="ANSI" />
 <layout first="18" last="18" modifiers="commonModifiers" mapSet="JIS" />
 <layout first="21" last="23" modifiers="commonModifiers" mapSet="JIS" />
 <layout first="30" last="30" modifiers="commonModifiers" mapSet="JIS" />
 <layout first="194" last="194" modifiers="commonModifiers" mapSet="JIS" />
 <layout first="197" last="197" modifiers="commonModifiers" mapSet="JIS" />
 <layout first="200" last="201" modifiers="commonModifiers" mapSet="JIS" />
 <layout first="206" last="207" modifiers="commonModifiers" mapSet="JIS" />
</layouts>
```

<modifierMap> Element

The <modifierMap> element defines the mapping of modifier key combinations to <keyMap> table numbers. The element contains one or more <keyMapSelect> elements, each of which correspond to one <keyMap> table.

There are two required attributes:

id	An arbitrary string, used to identify this <modifierMap> elsewhere (currently, only in the <layout> element). This identifier must be unique across all <modifierMap> elements.
defaultIndex	The table number to use for modifier key combinations which are not explicitly specified by any <modifier> element within the <modifierMap>.

<keyMapSelect> Element

This element defines the set(s) of modifier keys which cause a particular key mapping table to be selected. It contains one or more <modifier> elements, each of which specifies modifer key combinations. Every entry in the modifier mapping table which matches one of the specified modifier key combinations is filled in with this table number. The <keyMapSelect> elements are processed in sequence, so if two or more <keyMapSelect> elements specify the same combination(s) of modifier keys, the later ones will overwrite the earlier ones.

There is one required attribute:

mapIndex	A table number, starting from 0, to which modifier key combinations specified by <modifier> elements within this <keyMapSelect> element should be mapped. The table numbers should be contiguous and compact as the underlying implementation uses an array.

<modifier> Element

This element specifies one or more sets of modifier key combinations which should map to the table number specified in the enclosing <keyMapSelect> element. This element is empty in the XML sense (it's written as <modifier />). There is one required attribute:

keys	A string specifying one or more combinations of modifier keys. The string consists of one or more of the following separated by whitespace:

shift	The left Shift key
rightShift	The right Shift key
anyShift	Matches either the left or right Shift key
option	The left Option key
rightOption	The right Option key
anyOption	Matches either the left or right Option key
control	The left Control key
rightControl	The right Control key
anyControl	Any Control key
command	The Command key
caps	the Caps Lock key

The presence of a word means that the corresponding modifier must be pressed. The absence of a word means that the corresponding modifier must *not* be pressed. A word followed by ? indicates that the state of the modifier is irrelevant (it may be either pressed or not pressed). For example, `command?` means that this modifier element selects modifier key combinations with the command key either pressed or not pressed (i.e., the state of the command key is irrelevant).

An "any" modifier matches modifier key combinations where either or both of the left and right keys are pressed. For example, "`anyShift`" will match modifier key combinations where the left Shift key, the right Shift key, or both are pressed; it will not match combinations where neither is pressed. "`anyShift`?" will match all combinations of modifier keys where either or both of the Shift keys are either pressed or not.

Note that many hardware keyboards do not have both left and right versions of a modifier key. It is usually safest to specify modifers in terms of the "any" variants (`anyShift`, `anyOption`, `anyControl`).

```
<modifier keys="anyShift? anyOption caps?" />
```

This specification will match all modifier key combinations where either the left or right Option key is pressed (or both are pressed), and the command key and left and right Control keys are *not* pressed. The state of the Shift and Caps Lock keys are irrelevant.

```
<modifier keys="caps" />
<modifier keys="command" />
```

This specification will match all modifier key combinations where either caps lock key is down with no other modifiers, or the command key is down with no other modifiers.

```
<modifier keys="" />
<modifier keys="anyShift command caps?" >
<modifier keys="anyShift? command caps" />
```

This specification will match modifier key combinations where

- No modifier keys are down
- Either shift key and the command key are down, and control and option are up.
- The caps lock key and the command key are down, and control and option are up.

\<keyMapSet\> Element

This element collects a set of tables that are used to map from key presses to results. Typically there is one set for each class of hardware keyboard. Since the ISO keyboard used in Europe can be handled with the same layout as the ANSI keyboard used in the US, that leaves ANSI and JIS (used in Japan) as the two classes that need to be handled.

This element contains one or more `<keyMap>` elements, each of which specifies a particular mapping from virtual key codes to results. Typically there will be one such table for each relevant combination of modifier keys.

There is one required attribute:

id A string which serves as an identifier for this `keyMapSet` when referenced from elsewhere in the XML file. This identifier must be unique across all `keyMapSet` elements.

\<keyMap\> Element

This element specifies a particular mapping from virtual key codes to results. It contains one or more `<key>` elements which specify what to do when a particular virtual key code is received. The size of the resulting mapping table is determined by the largest virtual key code specified by any table in the `keyMapSet`; unspecified entries are filled with null results (they don't do anything, in other words). There are one required and two optional attributes:

index   The table number. This is referenced from the `<keyMapSelect>` element.

baseMapSet   The identifier of a base map set containing a `<keyMap>` element of which this element is a modification. `<key>` entries within the current element override entries from the specified base `<keyMap>` element. Virtual key codes not overridden are copied from the base element. If this attribute is present then the baseIndex attribute must also be present.

baseIndex   The table number of a `<keyMap>` element within the `<keyMapSet>` specified by baseMapSet. `<key>` entries within the current element override entries from the specified base `<keyMap>` element. Virtual key codes not overridden are copied from the base element. If this attribute is present then the baseMapSet attribute must also be present.

Typically the baseMapSet and baseIndex are used to provide overrides for a kind of hardware keyboard. The most frequent usage is to provide differences between the ANSI keyboard layout and the JIS keyboard layout, e.g.:

```
<keyMapSet id="JIS">
    <keyMap index="0" baseMapSet="ANSI" baseIndex="0">
    (overrides of ANSI layout for JIS keyboards)
```

`<key>` Element

This element specifies what to do when a particular virtual key code is received. The element may be empty or non-empty; it is only non-empty if an action is specified inline.

The following attribute is required:

code   The decimal number of the virtual key code which this `<key>` element maps. This number must be unique across all `<key>` elements within a particular `<keyMap>` element. (You can't map the same key in two different ways.)

It is intended that at some point named entities will be available to specify the key codes, but this is not yet implemented.

The `<key>` element takes one of three different forms:

1. `<key code="virtualkeycode" output="string" />`

   When the given virtual key code is received, the specified string of UTF-16 values is unconditionally output. The string must have at least one UTF-16 code point, and may contain any character legal in an XML attribute value. Literal characters are not limited to ASCII, and since XML files are themselves encoded in Unicode, many characters can be typed directly.

   Unicode scalar values may be specified using an XML numeric character entity, either in decimal or hexadecimal. Unicode scalar values outside Plane 0 are specified by a single numeric character entity, not two; this is converted to two UTF-16 values (a surrogate pair). Characters which are illegal in XML (< and ", and C0 control values among others) must be specified via a numeric character entity. Named entities are not supported.

   Example:

   `<key code="0" output=""Wow!→𠀋"" />`

   This `<key>` element specifies that when virtual key 0 ("a" on the US keyboard layout) is struck, the Unicode string:

   "Wow!

   followed by U+2192 RIGHTWARDS ARROW followed by U+2000B (a plane 2 ideograph, represented by a surrogate pair) followed by

   "

   is output. The entire UTF-16 string in hex is:

   201C 0057 006F 0077 0021 2192 D840 DC0B 201D

2. `<key code="virtualkeycode" action="name" />`

   When the given virtual key code is received, the named action is executed. Actions are specified via the `<actions>` element; see below.

3. `<key code="virtualkeycode">`
        `<action>`

```
            (an anonymous action)
        </action>
    </key>
```

When the given virtual key code is received, the single `<action>` element within the `<key>` element is executed. The result is the same as if the `<action>` element were specified within the `<actions>` element and referred to by name as in the second form, above. Note that anonymous actions specified in this way cannot be referenced by multiple `<keyMap>` tables, except through the copying mechanism (`baseMapSet` and `baseIndex`).

`<actions>` Element

The `<actions>` element is optional. There may be at most one per XML document. It contains one or more `<action>` elements, specifying named actions which may be referred to from the `<key>` elements.

`<action>` Element

The `<action>` element specifies a set of actions to take based on the current state of the state machine. This is specified by one or more `<when>` elements, each of which specifies one or more states, and an action to take when the action is executed and the current state matches. There is one attribute:

id	An arbitrary string identifying this action. It must be unique across all `<action>` elements.

The id attribute must be present if the `<action>` occurs within an `<actions>` element, and must not be present if the `<action>` is inside a `<key>` element.

`<when>` Element

This element specifies a particular result when the state machine is in a particlar state or set of states. There are several variants of this element; all are empty. The attributes are:

state	Required. This specifies the state the state machine must be in for the `<when>` element to apply. The state may be an arbitrary string, a decimal number, or the special string `"none"`, indicating the base (initial) state. If the `<when>` element specifies a range, this is the beginning of the range, in which case it must be a number. A `<when>` element specifying state "none" must be the first in the enclosing `<action>`.
through	Optional. The `<when>` element specifies a range of states rather than a single state. Must be a decimal number. If omitted, a single state is specified.
next	Optional. If present, specifies the next state to enter. If the `<when>` element specifies a range, must be a decimal number. Defaults to state none (i.e., return to the default, base state).
output	Optional. If present, the string to emit. If the `<when>` element specifies a range, must be a single UTF-16 code point. Defaults to no output.
multiplier	Optional. A decimal number between 1 and 255. The `<when>` element must specify a range. The difference between the input state and the start of the range (specified by the state attribute) is multiplied by this number, then added to the next state number and/or the output UTF-16 value.

The compiler will automatically assign numbers to named states; state numbers may also be referenced directly. The numbers assigned by the compiler to named states will be larger than any number referenced directly, so named states and numbered states are always distinct. If the highest state number referenced directly is large enough that there are not enough valid state numbers left to assign to the named states, an error occurs.

It's best to think about the `<when>` element in terms of two distinct forms:

1. `<when state="id" [next="id"] [output="string"] />`

   This form specifies an optional next state and an optional output string. One of them must be present. This is by far the most frequent form.

   See the example after the description of the `<terminators>` element to better understand this form.

2. `<when state="first" [through="last"] [multiplier="number"] [output="char"]`

```
[next="next"] />
```

This form specifies that when the input state is between first and last (inclusive), that the next state should be (input-first)*multiplier+next, and/or the output character should be (input-first)*multiplier+char.

This form is used for keyboards that generate a large range of characters with sequential values. Examples include the Unicode Hex keyboard, or a keyboard that generates Korean hangul.

See "Supporting Unicode Input" and the Unicode Hex keyboard layout XML file to better understand this form.

\<terminators\> Element

This element is optional, and may appear at most once in a file. It contains a series of \<when\> elements, which specify what to do when no action matches the current state. Each \<when\> element may specify only a single state and an output string. A next state may not be specified. If there is no \<when\> element for a particular state, the default is to return to the base (default) state.

A Complete Dead Key Example

Here is an excerpt from a keyboard layout that shows how to specify a dead key combination that handles e acute (é). This assumes a US keyboard layout.

In the \<keyMap\> for the unshifted keyboard, we have:

```
<key code="14" action="e" />
```

This specifies that when key code 14 is received, we take the action named "e".

In the \<keyMap\> for the option key, we have:

```
<key code="14" action="acute" />
```

This specifies that when key code 14 is received with the option key held down, we take the action named "acute".

```
<actions>
    <action id="acute">
        <when state="none" next="acute" />
    </action>
    <action id="e">
        <when state="none" output="e" />
        <when state="acute" output="é" />
    </action>
</actions>
<terminators>
    <when state="acute" output="´" />
</terminators>
```

If e is typed alone, we get e. If option-e is typed followed by e, we get é. If option-e is typed followed by any other key, we get ´ alone, followed by whatever that second key generates.

Note that we also could have specified the actions via:

Unshifted:

```
<key code="14">
    <action>
        <when state="none" output="e" />
        <when state="acute" output="é" />
    </action>
</key>
```

Option key:

```
<key code="14">
    <action>
        <when state="none" next="acute" />
    </action>
</key>
```

However, this approach can get cumbersome if the same actions need to appear in more than one place. Having all the actions collected in one place can help make the file more readable, too.

A complete example of a keyboard layout would be too lengthy for this tech note. See the Unicode Hex Input keyboard and other keyboards in `/System/Library/Keyboard\ Layouts/Unicode.bundle/Contents/Resources/` for complete examples.

Back to top

References

The Unicode Standard, Version 3.0. Addison-Wesley 2000, ISBN 0-201-61633-5.

The Unicode Consortium web site.

Supporting Unicode Input.

The Unicode hex input keyboard (and many other XML Unicode keyboards) can be found in:
`/System/Library/Keyboard\ Layouts/Unicode.bundle/Contents/Resources/`

Back to top

Downloadables

| | Acrobat version of this Note (72K) | Download |

Back to top