# SILKey

**The SIL Keyboard Manager System for Macintosh**
**Designed and implemented by Jonathan Kew**

Version 1.1 — 29 October 1996

Distributed by:
SIL International Publishing Services
7500 West Camp Wisdom Road
Dallas, TX 75236, U.S.A.

Address feedback and questions to the address above or:
Internet: fonts@sil.org
Phone: (972) 708–7495

## Copyright notice

## Introduction

SILKey is a suite of programs that can be used to modify the behavior of the Macintosh keyboard when typing in any standard Macintosh word processor or other text-editing program. It is possible to remap keys (make them generate characters other than the standard ones), to alter the sequence in which characters are entered into the document, to prevent illegal combinations being typed, to combine sequences of keystrokes into single characters, to generate multiple characters from a single keystroke, and other types of behavior.

The development of SILKey was inspired by the excellent KeyMan program for Windows (available from address above or `www.sil.org`); however, it is neither a port nor a clone of KeyMan. (There is no KeyMan code in SILKey, nor is it fully compatible with KeyMan keyboard files.)
SILKey should run on any Macintosh with any variety of System 7. It has been tested to some extent with various sub-versions of system 7.1 and 7.5. It has not been widely tested, however, and bugs and incompatibilities doubtless remain to be discovered!

The SILKey system has three components:

**SILKey Extension** is a small extension that allows SILKey to monitor and modify the key sequences being typed, no matter what application is in use. It must be installed in the Extensions folder for SILKey to work.

The **SILKey** application actually processes the keystrokes as you type, following the instructions in a keyboard file. SILKey must always be running (in the background) for a SILKey keyboard to be used; if you quit SILKey, the keyboard immediately reverts to the standard operating system behavior.

**Weaver** is the keyboard editor and compiler. It is used to create and modify keyboard files.

# Installation

Installing SILKey is straightforward:

1. Drag the SILKey Extension file to the icon of your (closed) System Folder. The Finder should automatically put it in the Extensions folder.

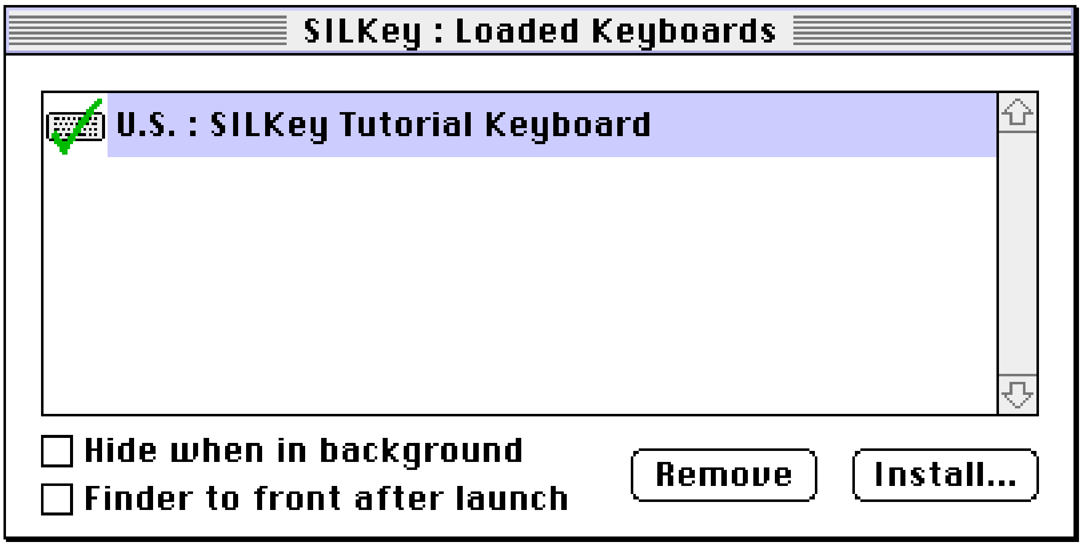
2. Drag the SILKey and Weaver applications to any convenient location on your hard disk (for example, an Applications or Utilities folder). If you want to try the sample keyboard definitions provided (*SILKey sample* and *Pig Latin kbd*), drag these to your hard disk as well.

3. Restart your Macintosh, in order to activate the SILKey Extension (which is installed during restart).

Note that holding down the mouse button during the startup process will prevent the SILKey Extension loading; if the extension is not loaded, the SILKey application will also refuse to run.

## Quick Start

Once SILKey is installed, you can easily try it out. The sample keyboards provided have already been ‘compiled’ (more on this later), so you can use them directly with SILKey. To try out the *SILKey sample* keyboard file, just launch SILKey (if you didn’t arrange for it to launch at startup). Bring it to the foreground (it launches ‘behind’ the Finder) by choosing it from the Application menu, or double-clicking the SILKey icon a second time. Click on the Install… button in SILKey’s window, and choose the *SILKey sample* file. It will be shown in the list of installed keyboards:


SILKey : Loaded Keyboards
U.S. : SILKey Tutorial Keyboard
Hide when in background
Finder to front after launch
Remove
Install...

The check mark on the keyboard icon indicates that the keyboard is active; clicking on this icon switches between active and inactive states, so you can temporarily disable the keyboard without having to actually remove and reload it.

(This sample keyboard definition is set up to work with the standard Macintosh U.S. keyboard, as indicated by “U.S.” in the SILKey keyboard list. If your system is set up with a foreign language keyboard, you’ll need to open the Keyboard control panel and choose the U.S. layout before this keyboard file will work.)

Once the sample keyboard file is loaded, launch a text-editing application. Now try typing the following key sequences:

```
'a `e ^i "o 'U `A ^E "I 'O `U
believe beleive receive recieve
???
```

Notice how SILKey modifies what you type; the sequences such as `'a` and `` `e`` become accented vowels, and the order of `ie`/`ei` is modified depending on the context. (Beware: this sample keyboard doesn’t really implement full English spelling rules for `ie` and `ei` combinations!)

# Tutorial

Let’s develop a simple SILKey keyboard definition from scratch. We’ll build a keyboard definition for use with standard Macintosh Roman fonts, which will provide alternative ways to keyboard some of the obscure option-key combinations.

To begin, launch SILKey and remove any loaded keyboard definitions (bring the SILKey window to the front, click on the keyboard name to select it, and click the Remove button). Launch Weaver, and choose New to create a new empty window.

## The basics

It is a good idea to include comments at the beginning of a keyboard definition file, giving details of what the keyboard is used for, who designed it, etc. So enter a line or two of relevant comments:

```
     // SILKey tutorial keyboard: JK, 23-Apr-95
     // Demonstrates simple SILKey rules.
```

First we need a header section for the keyboard definition. Assuming standard U.S. system software, we’ll set up this keyboard to work with the normal U.S. keyboard layout. (If you have non-U.S. software, you can check in the Keyboard control panel for the name of your keyboard layout.) Enter a line to specify this:

```
     keyboardname "U.S."
```

Then we also need to specify the rule group SILKey should use to begin processing each keystroke. Our keyboard will only need one rule group, which we’ll call Main:

```
     initialgroup(Main)
```

That’s all the header we need for now; next, we can start writing rules. Our first rule will be one to convert the sequence /= (slash followed by equals) into the character ≠ (the not-equals symbol). First we need to name the rule group, then we can enter the rule itself:

```
     group(Main) using keys
     "/" + "=" > "≠"  [As of SILKey 1.1, the + is optional, but we retain it for
clarity in the documentation of key-match rules]
```

(To type the ≠ character using the standard U.S. keyboard, press option-=.) This rule says that when the preceding context — the just-typed text — is a slash, and the current input character is an equals sign, this should all be replaced by a not-equals character.

Notice that the Compile and Install commands are unavailable; this is because we have never saved this file. So now choose Save As, and give the file a name (for example, SILKey Tutorial Keyboard).

To try out this rule, click the Install button (which automatically compiles the keyboard definition before asking SILKey to install it). The SILKey window should show that this keyboard file is loaded (you may have to choose SILKey from the application menu to see its window). Now switch to a word processing application, or make a new ‘scratch’ window in Weaver, and try typing /=. Note how when you type the slash, it appears normally, but when you type the equals sign, the slash is deleted and replaced by the ≠ character.

Now let’s add a few more rules. Go back to the keyboard definition window in Weaver, and add some more lines (no need to enter the comments; they’re just to save you searching for the characters in Key Caps!)

```
"<" + "=" > "≤"  // use option-comma to type ≤
">" + "=" > "≥"  // use option-period to type ≥
"<" + ">" > "≠"  // alternative to /=
```

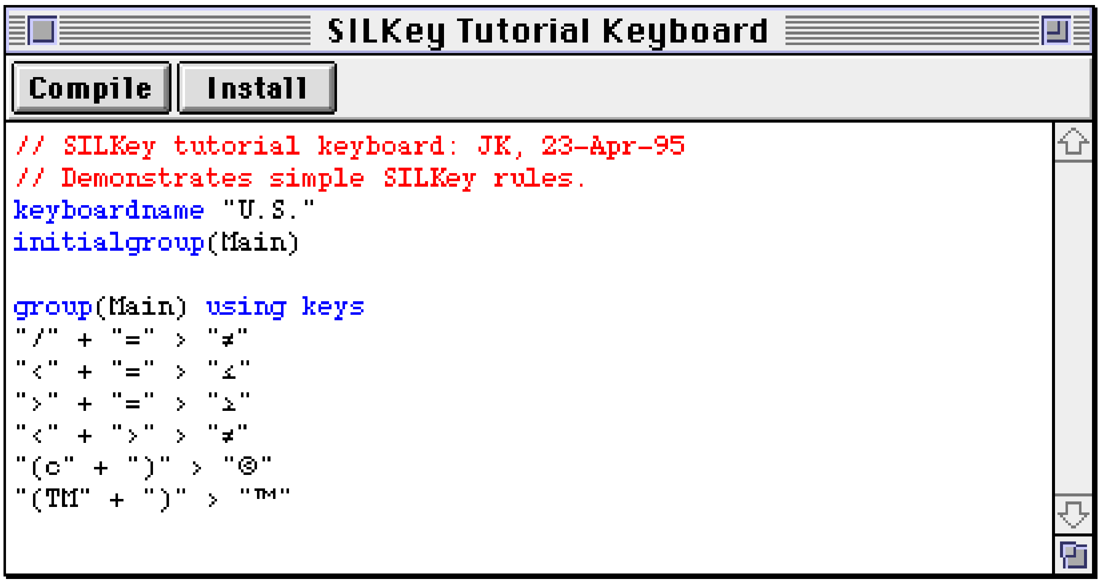
Reinstall the new modified keyboard (just clicking the Install button again is sufficient), and try out these rules.

How about a rule which recognizes more than just a pair of characters? No problem; the context can be more than one character:

```
"(c" + ")" > "©"    // option-g
"(TM" + ")" > "™"   // option-2
```

Again, click Install to update the keyboard loaded in SILKey, and try these out.

So far, our complete keyboard file looks like this in Weaver:



```
// SILKey tutorial keyboard: JK, 23-Apr-95
// Demonstrates simple SILKey rules.
keyboardname "U.S."
initialgroup(Main)

group(Main) using keys
"/" + "=" > "≠"
"<" + "=" > "≤"
">" + "=" > "≥"
"<" + ">" > "≠"
"(c" + ")" > "©"
"(TM" + ")" > "™"
```

By the way, if you have a color screen, then Weaver should color various parts of the keyboard definition when it compiles it, to aid readability. If you don’t have color, you can choose Preferences from the Edit menu, and opt to use different styles instead.

If Weaver encounters errors in the keyboard definition, it stops compiling and displays an error message in a dialog box. After you click OK to dismiss the dialog, the insertion point in the text window is left immediately following the item (a misspelled keyword, or whatever) that prompted the error message. (Try introducing a ‘typing error’, and see what happens when you recompile.)

## The any() and index() commands

Now we’ll do some accented vowels. Rather than using dead keys typed before the vowel, as in the standard Macintosh keyboard, we’ll implement modifiers that are typed after the vowel. First, we’ll make slash after a vowel produce an acute accent:

```
"a" + "/" > "á"  // option-e a
"e" + "/" > "é"  // option-e e
"i" + "/" > "í"  // etc...
```

This works, but it gets cumbersome to list dozens of similar combinations like this. So delete these rules (try them out first!), and we’ll use some more advanced features to make the keyboard definition simpler. First, in the header section (before the **group** line), define a couple of groups of characters using the **store** statement:

```
store(vowel) "aeiouAEIOU"
store(vacute) "áéíóúÁÉÍÓÚ"
```

Then, at the end of the rules, add a single rule like this:

```
any(vowel) + "/" > index(vacute,1)
```

This says that when the context is any character from the *vowel* store, and the input character is a slash, replace these with the corresponding character from the *vacute* store. By adding stores *vgrave*, *vcircumflex*, and *vdieresis*, and similar rules, you can easily provide convenient access to a whole range of accented vowels.

One problem with this, however, is that if you actually need the sequence a/ in a document, it will be difficult to type. For occasional use, you can work around the problem by typing it in an unnatural way: for example, type the slash first, then press the left arrow and type the *a*. Or type the *a*, then press the left arrow and then the right arrow (which causes SILKey to ‘forget’ the context; in general, anything that interrupts a typing sequence does this), then type the slash. But for characters which might be wanted quite often, some careful planning of keying sequences may be needed to ensure that they can easily be accessed. (See the SILKey sample keyboard for an alternative approach to accented vowels which may be more useful in practice.)

In the accented vowel rules we just looked at, the **index** command refers by name to the store from which a character is to be output; but what is the number ‘1’ for? This is because in a complex rule there might be multiple **any** commands in the context and key match parts, and multiple **index** commands in the output. This number links the **index** command to an **any** command on the left of the rule, where the **any** commands are regarded as numbered from 1 to however many are present. So, for example, the rule:

```
any(vowel) any(vowel) + "x" > index(vowel,2) index(vowel,1)
```

has the effect of reversing any pair of vowels if *x* is typed immediately afterwards. (Try it: add this rule to your sample keyboard and reinstall, then try typing *x* after a pair of vowels.)

While we have this (rather pointless) rule, we can try another feature, too. Notice that if you type a pair of vowels, then an *x*, the result is still a pair of vowels. Therefore, typing a second *x* will simply reverse them again, as the same rule will apply on the new keystroke as well. Suppose we don't want this; once the vowel-reversal rule has been executed, we want to consider the vowels 'finalized', so they will not be further modified by anything typed afterwards. To do this, we can add the **fix** command to the end of the rule, making it:

```
any(vowel) any(vowel) + "x" > index(vowel,2) index(vowel,1) fix
```

This command causes SILKey to 'forget' what it has output up to this point, so the reversed vowel pair will no longer be recognized as matching the context when the next keystroke is processed. With this form of the rule, typing a vowel pair followed by *x* still reverses the vowels, but then typing another *x* simply inserts the *x*, as the rule does not 'see' the appropriate context to apply again.

## Multiple groups and context-only rules

All our rules so far have been 'key-match' rules: they match the character typed by the user (the one after the + in the rule) provided the preceding context also matches. [As of SILKey 1.1, the plus is optional.] Each keystroke can only be matched in this way once; after a rule has matched the typed character (and therefore replaced it with its output), no other key-match rule can apply. However, it is sometimes helpful to apply additional rules after the initial matching of the keystroke. (It is hard to come up with a sensible English example for this, but in some complex orthographies it may be very useful.) This can be done with *context-only* rules, which operate only on the already-output text.

To try some context-only rules, we'll first create a new rule group, which will always be executed after the Main group. At the end of the file, begin a new group and enter a context-only rule, as here:

```
group(Trademark)
"SILKey" > context "™"
```

Note that there is no + in this rule; the left-hand side is all context, no key. Note also that the **group** line does not specify **using keys**, as our Main group did; this group will have context-only rules rather than key-match rules. The **context** command at the beginning of the output tells SILKey to preserve the context, rather than replacing it with the output; thus, the effect of this rule is to add ™ whenever the name *SILKey* is typed. (This is not really appropriate!)

Try this; it doesn't work! This is because we haven't done anything to make SILKey execute the rules in this group; it processes the initial rule group, then terminates. To 'call' a second group, we use the **use** command. This can be used anywhere in the output of any rule; for this purpose, the easiest way to cause our new group to be used is to add a nomatch rule at the end of the Main group:

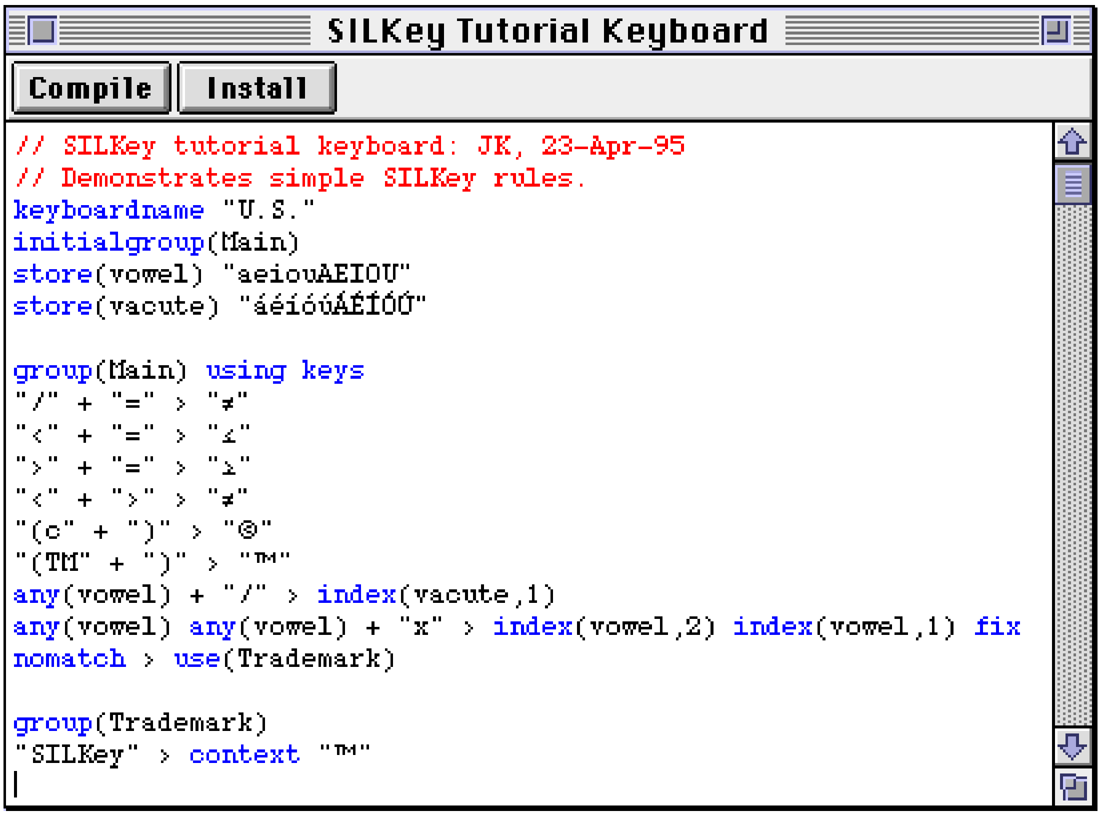
```
nomatch > use(Trademark)
```

This says that if no rule in the group has matched, then the Trademark group should be used. We could also add a match rule:

```
match > use(Trademark)
```

so that if any rule has applied, the second group will still be used as well. Now try the keyboard again; typing SILKey should now automatically produce a trademark symbol after the name. (Note that if you then try to delete it with the backspace key, it keeps reappearing; once it is deleted, the context again matches this rule, so it is reinserted! To delete it, you have to do something such as type an arrow key or click the mouse, to break the typing sequence.)

Our finished keyboard definition file should now look something like this:



```
// SILKey tutorial keyboard: JK, 23-Apr-95
// Demonstrates simple SILKey rules.
keyboardname "U.S."
initialgroup(Main)
store(vowel) "aeiouAEIOU"
store(vacute) "áéíóúÁÉÍÓÚ"

group(Main) using keys
"/" + "=" > "≠"
"<" + "=" > "≤"
">" + "=" > "≥"
"<" + ">" > "≠"
"(c" + ")" > "©"
"(TM" + ")" > "™"
any(vowel) + "/" > index(vacute,1)
any(vowel) any(vowel) + "x" > index(vowel,2) index(vowel,1) fix
nomatch > use(Trademark)

group(Trademark)
"SILKey" > context "™"
```

Of course, you may have added other similar rules as you went along; feel free to experiment!

If you want to print out a keyboard definition file, you will notice that Weaver lacks the standard Page Setup and Print commands in its File menu. However, you can open the file with any word processor or text editor, and print from there. Programs such as SimpleText which recognize “styled text” files will retain Weaver’s syntax coloring or styling, allowing you to print with highlighted keywords.

# Theory of operation

This section describes how SILKey works, and how it interacts with other parts of the Macintosh system. It may be helpful when considering how to implement complex keyboarding systems.

## Keyboard layout resources

When a key is pressed on the Macintosh keyboard, the system software uses a keyboard layout (KCHR) resource in the System file to map the key code generated by the keyboard to a character code which is passed to the active application. Non-U.S. Macintosh systems generally include both the U.S. KCHR and a national-language one implementing a country-specific layout. When installing System 7.5, there is an option to include 'international support'; if this is chosen, around 20 different KCHRs for various keyboard layouts are installed.

When multiple KCHRs are installed, the Keyboard control panel allows the user to choose which one to use. This choice is persistent; it is recorded in the System file, and the chosen layout remains in effect across restarts. There is also a system menu which may be displayed (between the application and help menus), allowing keyboard layouts to be changed at any time without going to the control panel. Keyboard changes from the menu are temporary; at the next restart, the system defaults again to the keyboard chosen in the control panel.

The keyboard menu is not normally displayed except when a non-Roman script (such as Arabic, Hebrew, Japanese, etc.) is installed. However, when working with multiple keyboard layouts, even without non-Roman scripts, it is helpful to display the menu. Weaver includes an option in its Preferences dialog (choose Preferences from the Edit menu) to enable the menu even on Roman-only systems.

It is important to be aware of KCHR resources for several reasons:

• Many simple rearrangements are better done with a new KCHR resource than with a program like SILKey.

• SILKey does not replace the KCHR's key-to-character mapping; rather, it operates after this mapping has been performed, on the character codes that result.

• A SILKey keyboard definition works with one specific KCHR, and to use multiple SILKey definitions (without constantly going to the SILKey status window to switch them on and off) requires multiple KCHRs.

The KCHR resource provides a built-in, standard mechanism for altering the key-to-character mapping of the Macintosh keyboard. Any keystroke may be mapped to any character code; it also allows 'dead keys' which combine with the following keystroke to access a different character. An example of this is the *option-e* keystroke (assuming the standard U.S. KCHR), which combines with a following vowel key to give an accented vowel. Thus, a system such as SILKey is not needed, and should not be used, to implement simple keyboard rearrangements or dead-key systems; for these, a custom KCHR provides a safer, more efficient, and more compatible solution. The ResEdit program can be used to edit KCHR resources fairly easily; it provides a visual editor where characters from a grid are dragged onto a keyboard layout. See any good ResEdit book or the manual for more information.

Each SILKey keyboard definition specifies (with the **keyboardname** statement) the name of the `KCHR` it is associated with, and only operates when this `KCHR` is active. Thus, to use multiple SILKey layouts, and switch among them easily, it is best to install one `KCHR` per layout (even if the `KCHR`s are all identical except in name). Then several SILKey keyboard definitions can be loaded simultaneously, and choosing a keyboard (`KCHR`) from the system keyboard menu will automatically cause the corresponding SILKey definition to be active. (Note that SILKey allows multiple keyboard definitions - limited only by available memory - to be loaded; however, only one keyboard definition per `KCHR` can be active at any time. Keyboards can be activated and deactivated in the SILKey status window.)

To simplify adding new names to the system keyboard menu, so that corresponding SILKey definitions can be used, Weaver provides a New Keyboard Resource command (in the File menu). This command creates a new `KCHR` resource with a name you give, and puts it in a standalone file (with the same name). To install it in the system, you just need to quit all applications, and drag the file onto the System Folder icon. A keyboard layout resource created in this way is an exact copy of the standard U.S. layout, and has no icon; the keyboard menu will show a default keyboard icon for it. To modify it (before installing) to create new dead keys, remove the standard dead keys, and so on, you need to edit it with ResEdit; to add a custom icon (like the flags used for Apple’s standard layouts), you need to add keyboard icon resources (types `kcs#`, `kcs4` and `kcs8`). Weaver does not perform these operations.

## How SILKey processes keystrokes

When a key is pressed, the system software uses the currently active KCHR to map the key to a character code. Normally, this code is then passed on (in a “key-down event”) to the foreground application. If SILKey is running, however, it intercepts and examines the key-down event before it reaches the application, and (assuming there is a keyboard definition loaded and active that corresponds to the active KCHR) it may modify the event (or events) that reach the application. At this point, there is no way to intercept the keystrokes *before* the KCHR maps the key to its corresponding character code. SIL's KeyMan utility, on the other hand, is able to do this.

There are two main types of rules in SILKey keyboard definitions: key-match and context-only. A key-match rule matches the character typed, possibly in a particular context (i.e., when the previously-typed text matches a certain pattern). When a key-match rule is executed, the input character is deleted and replaced with the output of the rule. Only one key-match rule can be applied to any given keystroke; once the input character has been ‘consumed’ by a rule that matches it, no subsequent key-match rule can possibly match it. A context-only rule, on the other hand, ignores the input character, and matches and modifies only the previously-typed material; subsequent context-only rules (if **use** commands call other rule groups) may match the new, modified context and modify it further. In addition, there are match and no-match rules, which ‘match’ or not depending whether any earlier rule in the group has matched; these are sometimes useful for ‘default’ behavior.

When it begins processing a keystroke, SILKey always starts with the rules in the initial rule group of the keyboard definition (as specified by the **initialgroup** statement). It works sequentially through the rules, checking each to see if it matches the input character in the current context. If it finds a key-match or context-only rule that applies, it executes this rule, and ignores all following key-match and context-only rules in the group; thus, only the first applicable rule in the group will be executed. If it finds a match rule at any point, and a rule (other than match or no-match) in the group has been executed, this rule is also executed; similarly for no-match rules. Thus, all match and no-match rules in the group are considered for execution in their turn; however, the most common use of these rules is probably just to have one of each (or only one) at the end of a group.

If the output of a rule includes a **use** command, SILKey suspends the current rule at that point and tries all the rules in the specified group. Any output already generated becomes part of the context for the rules in the called group; any output following the **use** command will follow output from any rules executed from the called group.

When SILKey starts processing a group, if the keystroke has not yet been matched by a key-match rule, and if the group is not a **using keys** group, then the new keystroke is passed on to the active application and becomes part of the context. After this has happened, it is not possible for a key-match rule (in this or any other group) to match on this input character; it is now context, not fresh input.

# Keyboard definition reference

A SILKey keyboard definition to be compiled by Weaver consists of a series of statements, each terminated by a *return* character. Thus, each statement is logically a single line, although if the line is longer than the window width in Weaver it may be wrapped onto several lines for display. A single logical line may also be broken onto multiple lines by typing a backslash character, **\**, immediately before the *return*; in this case, both the backslash and the *return* itself are ignored.

The keyboard definition is made up of header information, followed by one or more rule groups. Comments may also be included in the keyboard definition file; they are introduced by two slashes, `//` or **`c`**, and continue until the end of the line.

In the statement explanations that follow, **bold** shows the actual keywords Weaver looks for, while *italics* are used for placeholders in the statement syntax where specific items should be substituted. Square brackets, [...], are used to indicate optional items.

## Header

There are only three statements which are used in the header section:

**keyboardname** *string*
Specifies the name of the system keyboard layout (KCHR) this keyboard definition is to be used with. The keyboard definition will only be active when this is the current keyboard layout (as specified with the Keyboard control panel or the Keyboard menu).

**initialgroup(***name***)   begin  > use (***name***)**
Specifies which of the rule groups in the keyboard definition is used to begin processing. If this statement is missing, SILKey uses the first (or only) rule group as the initial group, but (at least in multi-group keyboard definitions) it is better style to explicitly state where processing begins.

**store(***name***)** *string*
Create a store with the given name, and store the characters in the given string in it. Multiple strings may be given, in which case they are concatenated; numeric character codes can also be specified. The **outs** command (see below) may also be used, to include the contents of a previously defined store in the new store without explicitly repeating it. The order in which characters are stored may be important, if the store will be used with the **index** command.

A *string* is a sequence of zero or more characters within quotes (either single or double quotes, but not curly or ‘smart’ quotes). There is no mechanism to escape special characters within a string; thus, if a string is to contain a single quote mark, it must be delimited with double quotes, and vice versa. A single string cannot contain both single and double quotes; however, wherever this might be useful, it is possible to use two strings in succession instead, one with each kind of quotes.

A *character code* is simply the numeric code for a character (such as an ASCII code number, or equivalent for fonts with other character sets). It is equivalent to a single-character string, but may be useful for non-printable characters. Character codes may be specified as decimal numbers (in the range 0–255), or in octal by prefixing the number with the letter *o* (e.g., o101, equivalent to decimal 65, or ASCII ‘A’), or in hexadecimal by prefixing the number with the letter *x* (e.g., x41, also equivalent to decimal 65). For better compatibility with the KeyMan for Windows, Weaver also allows a decimal character code to be prefixed with the letter *d*; note, however, that unmarked numbers are also treated as decimal, whereas KeyMan interprets them as octal.

A few character codes also have names which may be used instead for better readability:

**del**
the *backspace* character (generated by the *delete* key), decimal 8

**tab**
the *tab* character, decimal 9

**nl**
the *return* or *newline* character, decimal 13

A *name* is an identifier for a store or a group, and consists of one or more characters, not including whitespace or other characters (such as quote marks and parentheses) which have special meaning in the syntax. It may not begin with a number. Normally, names are made up of letters, perhaps with numbers to distinguish similar names in a set, though other punctuation characters may be used as well. Names are case-sensitive.

## Rules

The rules in a keyboard definition must be in rule groups; even in a simple keyboard, there must be a group header statement before the rules. The group header has the form:

**group(***name***)** [ **using keys** ]
Begin a rule group with the given name. If the optional **using key**s phrase is present, rules in the group will match on the keystroke typed by the user. It is not possible to include context-only rules in a **using key**s group. Because the + is optional, the rules will always be interpreted as key-match rules. Key-match rules will not work at all in a group without **using key**s. Do not define more than one **using key**s group in a keyboard definition.

There are four types of rules: key-match, context-only, match, and no-match. When processing a keystroke, SILKey tries each rule in the active group in turn. In any group, only the first match that applies is executed. If a match rule is encountered, and an earlier key-match or context-only rule in the group has been executed, the match rule is executed; similarly, a no-match rule is executed if no earlier key-match or context-only rule has been executed.

Here are the general forms of the four types of rules:

**key-match rule**
[ *context* **+**] *keystroke* > *output*

**context-only rule**
*context* > *output*

**match rule**

**match** > *output*

**no-match rule**
**nomatch** > *output*

In these rule forms, *context* is made up of a sequence of quoted strings, character codes, and **any** commands. The **any** command has the form:

**any(***store-name***)**
Matches any of the characters in the contents of the given store; which character was matched is remembered for use by an **index** command in the output part of the rule.

A *keystroke*, as used in the key-match rule, may be a single-character string, a numeric character code (or one of the character names defined above such as **tab**), or an **any** command.

Another kind of key-match rule involves deadkeys. You must define at least two rules for deadkeys. The first rule assigns the deadkey. The second (and any subsequent rules) make use of it.

*keystroke* > **deadkey**(*#*)[assigns a deadkey]

**deadkey**(*#*) [+]*keystroke* **>** *output* [uses a deadkey which has been assigned] or
**dk**(*#*) [+]*keystroke* **>** *output* [shorthand for "**deadkey**"]

A # represents a number from 1-255.

*Deadkeys* can be assigned in the using keys group. If the user types one of these deadkeys, nothing will happen on the screen, but SILKey will wait for the next keystroke to determine what to do. This will be determined by a second rule using the word **deadkey** in the context (left-hand side), plus the keystroke, and giving the desired output.

The *output* of a rule is a sequence of strings, character codes, and any of several commands:

**context**
Note that this is a **command**. It is not the same as the "*context* "defined above. It means: output whatever was matched by the *context* part of the rule. (If **context** occurs as the first item in the *output* of a rule, the matched context is left untouched; otherwise, it is deleted before the output is generated. The **context** command may be used elsewhere in the output, if required, in which case the deleted context is re-generated when required.)

**outs**(*store-name*)
Output the contents of the specified store. This is equivalent to a literal string with the same contents as the store; however, if a particular string is to be output by many rules, putting it in a store with a suitable name and using **outs** may make the keyboard definition easier to understand.

**index**(*store-name*,*any-index*)
*any-index* is a number referring to an **any** command on the match side of the rule, in the range 1 to the number of **any** commands used. The **index** command outputs the character

from store *store-name* which is in the same position in the store as the character which matched in the **any** command's store. An example may make this clearer:

```
store(vowel) 'aeiou'
store(vacute) 'áéíóú'

group(Main) using keys
any(vowel) + '/' > index(vacute,1)
```

Here, when the rule matches, the **index** command outputs the character from the *vacute* store which corresponds to the character which matched in the **any** command. Thus, if the user types *e*, the second character in *vowel*, and then a slash, the result is the second character in *vacute*, namely *é*.

**beep**
Plays the system beep sound. This may be used to indicate an illegal key sequence, for example.

**key**
This is used on the output side of the equation to copy the keystroke into the context. Previously, you had to repeat the description, which could be lengthy**.**

**fix**
Causes SILKey to 'forget' (but not delete) everything which has been output so far, so that it cannot be matched as context of a subsequent rule (either later in the processing of the current keystroke or on a later keystroke). This is useful to 'fix' the output of a rule, when it might otherwise be modified in an undesired way by a later rule.

**use**(*group-name*)
Try the rules in group *group-name*. As noted above, SILKey will only execute the first key-match or context-only rule in the group that applies; it will also execute match and no-match rules as appropriate. When processing of the called group ends, SILKey continues with any more output that follows the **use** command.

**return**
Immediately terminates processing of the current keystroke. Anything following **return** will be ignored, and any unfinished output from rules which called the current group with a **use** command will not be completed. Control returns to the user, and SILKey awaits another keystroke.

# KeyMan Compatibility

To simplify the conversion of KeyMan control files for SILKey, several KeyMan statements which have no SILKey equivalents are ignored (rather than generating error messages) by Weaver. These are the **NAME**, **VERSION**, **HOTKEY**, and **BITMAPS** statements. Weaver regards any line beginning with one of these keywords as a comment line.

When opening a file, Weaver will remove any line feed characters which immediately follow carriage returns, to deal with the difference between DOS and Macintosh line-end conventions. (Note that it will not re-insert line feeds when saving the file. If a Weaver-edited file is to be moved back to DOS, this may need to be done with some other program.)

**c, $, //**
The character **c** or **$** may be used (in place of Weaver's preferred double slash, //) to begin comments.

**+**
The optional plus sign in KeyMan is now optional in SILKey as well. Originally it was used to set off the context from the current keystroke in SILKey.

**,**
You may now separate items in the output with commas, as is possible in KeyMan.

**$Weaver:**
Use this prefix on any SILKey-specific line in your keyboard file. The line will then be performed as normal in SILKey, but treated as a comment in KeyMan. Conversely, command lines specific to KeyMan may now be preceded by "**$KeyMan:**". **$Weaver:** is only implemented in KeyMan 3.2 or higher versions.

**KeyMan-index**
This resolves the difference in syntax for indexes between KeyMan and SILKey. SILKey will automatically "convert" the KeyMan index syntax to work appropriately in SILKey.

**octal-numbers**
This causes unmarked numbers (no 'd', 'x', or 'o' prefix) to be interpreted as octal when used as character codes (though not when used as index() offsets), which is the KeyMan behavior.

**ignore-case**
This causes SILKey to ignore upper and lower case differences in variable names, which is how KeyMan handles them. Keywords, such as commands, however, are never case-sensitive in SILKey or KeyMan.

**KeyMan-compatible**
This has the effect **of KeyMan-index, octal-numbers**, **and ignore-case** put together.

## Other Notes on KeyMan compatibility

### The following KeyMan function is not implemented in SILKey:

Virtual key input

However, you may be able to mimic **virtual key input** using the following guidelines.

If the KeyMan file makes use of “virtual keys” on the left side of the equation, you will need to make some adjustments in order to use one file for both systems. You may also be able to have the same keyboard layout. Here are tips to help you keep your PC and Mac keyboards identical.

On the KeyMan side, use only keys which are found on the main keyboard, not the numeric keypad or extended keys.

On the KeyMan side, do not use combinations involving the control keys [LCTRL] or [RCTRL].

The ALT key will be used in KeyMan for the OPTION key in SILKey. You may use the SHIFT key in combination with these.

In KeyMan, you may use the following equation to have ALT-SHIFT-A produce character code 65 “A”.

```
        + [ALT SHIFT K_A] > "A"
```

A simple way to do this in SILKey, without using character codes, is to simply type the sequence in. The same example as above would look like this:

```
        $WEAVER:  + "Å" > "A"
```

Another way to get this same key sequence to produce “A”.in SILKey is to find which character code is normally produced on a Mac (using the keyboard you selected above), when you type `OPTION SHIFT` and “A”.

The equivalent equation, therefore, to be used in SILKey is:

```
        + d129 > "A"
```

You must give the *character code* on the left side of the equation.

Note that not all sequences produce a single character code on the Mac. For instance, there is no `OPTION-E` or `OPTION-I`. (These are dead keys which require something additional to be typed before displaying. This is, of course, dependent on which keyboard or KCHR is in use.)

## Infinite Loops and Recursion

KeyMan will quit re-iterating or recursing should you accidentally code an infinite loop. SILKey will stay in the infinite loop and hang your machine. Recursion is legal in both, but be careful in using it, since KeyMan doesn’t let you know and SILKey only hangs when you are using the keyboard, not at the compile.

## Backspace

If you wish to use [K_BKSP], copy each line where this occurs and change all occurrences of [K_BKSP] to **del**. Preface each of these new lines with **$Weaver:**. Preface all original lines that have [`K_BKSP`] with **$KeyMan:**.

## Final Comments

It will often not be possible to use a KeyMan keyboard definition in SILKey with no modifications, for several reasons. The systems do not support exactly the same set of features. In many cases, the desired output will be different because of font differences between PC and Macintosh fonts. It is also not guaranteed that the behavior of complex keyboard definitions will be the same; the author of SILKey does not know enough detail about the behavior of KeyMan to determine this. In particular, KeyMan performs the longest rule first (the one with the highest number of characters defined on the left side of the equation), whereas SILKey performs rules in the order they appear.

This document attempts to describe SILKey's operation in some detail; it is for the would-be keyboard 'porter' to determine if the results will be appropriate for a keyboard definition originally written for KeyMan.