

Language change summary from version 3.x to 4.0

For full details on any of the changes, see the appropriate reference page.

- BITMAPS header statement is now BITMAP; only one bitmap is required.
- VERSION 3.2 header statement is now VERSION 4.0.
- A new header statement, LANGUAGE, is required; see language codes on page 2 for details.
- Other optional new header statements are: LAYOUT, COPYRIGHT and MESSAGE.
- The plus (+) character is now required in any key-based rules (`context + key > output`).

Language Reference

This language reference describes the keyboard programming language of Tavultesoft Keyboard Manager.

You can check for the latest version of this document at <http://www.tavultesoft.com/keyman/docs/>.

The layout of a keyboard file is organized in two distinct parts: the header, and the body of the code.

Header

The header consists of statements that provide information about the keyboard: the name, version of Keyman it was created for, hotkeys, and title bar icons. The header must come at the start of the file. The statements should be entered uppercase so as to distinguish them from statements in the main body of the code; however, Keyman will recognize them anyway.

Body

The body of the keyboard can contain *stores* and *groups* of rules.

Stores are used to keep a table of keys which can be referenced to a second table of output characters. *Rules* are the heart of a keyboard file. They describe the action Keyman should take when processing a key combination. They can be dependent on the context of characters before them and produce any characters that you wish.

Rules are placed in *groups*. Each group can contain one or more rules; a group is similar in many ways to a subroutine or procedure in Visual Basic. Most keyboards will not need multiple groups. See the **use** statement for more information about groups.

Stores are described in more detail in the **store** statement reference.

Each rule consists of three parts: the context, keystroke, and output. Either the context or the keystroke are optional in some situations. The context is what is compared to characters already on the screen. The keystroke is compared to the key you type, and the output is what will replace and supplement the context on the screen.

Keyman has a buffer for the screen characters of 64 bytes. The length of the context and the output is by default 16 bytes for each. These limits can be set in the [Advanced] section of KEYMAN.INI. See the Keyman User's Manual for more information on KEYMAN.INI.

Rules can have an optional context. The base context is the characters that were output to the screen after Keyman translated them. The base context is usually 64 characters long and the rule context is usually 16 characters long, although both are modifiable. You can compare the rule context to the base context; if it matches (and the key too), that rule will be used in the output of the new string. The context, output and keystroke are specified in ExtendedString format.

The three parts of a rule (context, key, and output) are put together in a style similar to SIL CC:

Context + Key > Output

The ' +' is an optional character; it is just supplied to make it easier to see the break between context and key. **Note: The plus character ('+') may be required in later versions of Keyman.** The simplest type of rule is simply one-to-one key mapping. The most complex can have a table of keys which can be referenced in many different ways to match the context.

Variable Types

The different types of variables/constants and the prefixes usually used when describing them are:

<i>TextString</i> (ts...)	A string of text enclosed by double quotes
<i>StoreName</i> (sn...)	The name of a store in that file (no quotes)
<i>Number</i> (n)	A number such as an offset
<i>ExtendedString</i> (xs)	A string that can have "", ", d..., x...,
<i>Identifier</i> (i)	A string not enclosed by quotes; file names.

ExtendedString/Char format

The ExtendedString and ExtendedChar formats are strings/characters that can be written as a quoted string and/or decimal/hexadecimal/octal codes. An extended string can be made up of any amount of these different codes. There are five ways of representing any character in the string; these are shown in the table below:

Code	Description	Example
'A'	In single quotes (you can represent a double quote character (") inside single quotes)	+ 'C' > 'X'
"A"	In double quotes (you can represent a single quote character (') inside double quotes)	+ " " > ' ' "
d65	As a decimal (useful for upper-ascii numbers and codes like optional hyphen (d31).	+ d66 > d74
x41	As a hexadecimal (base 16) code (mostly useful for people used to programming with hexadecimal numbers)	+ x50 > x88
101	As an octal (base 8) code (to provide compatibility with SIL-CC)	+ 124 > 204

The extended string format can also include statements such as **any** and **index** that will be converted and/or expanded to the correct sequences in memory when the keyboard is loaded.

Comments

A comment can be inserted in a line by preceding it with a 'c' identifier. The identifier must be preceded and followed with a space character. The comment continues until the end of the line.

Language Codes

Windows 95 and NT have a standard definition for languages that Keyman 4 integrates with. Each language is given a code (shown in the table below), and dialects of this language are given sub-codes. The standard defined languages are shown below; for other, undefined languages, either use a new sub-language code in an existing, related language, or use the user-defined language codes x200 to x3ff and sub-language codes x20 through x3f.

User-defined language codes

If you decide to use a user-defined code, you can request a unique code for your language at <http://www.tavultesoft.com/keyman/langcode/>. Tavultesoft will then keep track of the languages by assigning codes to ensure that Keyman keyboards will not conflict with each other. If you do not plan to use the keyboard in conjunction with any others, or to distribute it to other people, it is safe to use any language code you wish.

Language and sub-language codes

x36 Afrikaans	x01 South Africa	x02 Iraq
x1c Albanian	x01 Albania	x03 Egypt
x01 Arabic	x01 Saudi Arabia	x04 Libya
		x05 Algeria

	x06 Morocco		x02* Irish
	x07 Tunisia	x07 German	x01 Germany
	x08 Oman		x02 Switzerland
	x09 Yemen		x03 Austria
	x0a Syria		x04 Luxembourg
	x0b Jordan		x05 Liechtenstein
	x0c Lebanon	x08 Greek	x01 Greece
	x0d Kuwait	x0d Hebrew	x01 Israel
	x0e United Arab Emirates	x39 Hindi	x01* Hindi
	x0f Bahrain	x0e Hungarian	x01 Hungary
	x10 Qatar	x0f Icelandic	x01 Iceland
x2d Basque	x01 Spain	x21 Indonesian	x01 Indonesia
x23 Belarussian	x01 Belarus	x10 Italian	x01 Italy
x02 Bulgarian	x01 Bulgaria		x02 Switzerland
x03 Catalan	x01 Spain	x11 Japanese	x01 Japan
x04 Chinese	x01 Taiwan	x2c Kampuchean	x01 Cambodia
	x02 PRC	x12 Korean	x01 (Extended Wansung) -
	x03 Hong Kong		Korea
	x04 Singapore		x02 (Johab) - Korea
	x05* Macao	x2b Laotian	x01 Laos
x1a Croatian	x01 Croatia	x26 Latvian	x01 Latvia
	x02 Serbia (Latin)	x27 Lithuanian	x01 Lithuania
	x03 Serbia (Cyrillic)	x2f Macedonian	x01* Macedonian
x05 Czech	x01 Czech Republic	x3e Malay	x01* Malaysian
x06 Danish	x01 Denmark		x02* Brunei
x13 Dutch	x01 Netherlands	x3a Maltese	x01* Maltese
	x02 Belgium	x28 Maori	x01 New Zealand
x09 English	x01 United States	x14 Norwegian	x01 Norway (Bokmal)
	x02 United Kingdom		x02 Norway (Nynorsk)
	x03 Australia	x15 Polish	x01 Poland
	x04 Canada	x16 Portuguese	x01 Brazil
	x05 New Zealand		x02 Portugal
	x06 Ireland	x17* Rhaeto-Romanic	x01* Rhaeto-Romanic
	x07 South Africa	x18 Romanian	x01 Romania
	x08 Jamaica		x02* Moldavia
	x09 Caribbean	x19 Russian	x01 Russia
	x0a Belize		x02* Moldavia
	x0b Trinidad	x3b Saami	x01* Saami (Lappish)
	x0c* Zimbabwe	x1b Slovak	x01 Slovakia
	x0d* Philippines	x24 Slovene	x01 Slovenia
x25 Estonian	x01 Estonia	x2e Sorbian	x01* Sorbian
x38 Faeroese	x01 Faeroe Islands	x0a Spanish	x01 Spain (Traditional
x29 Farsi	x01 Iran		Sort)
x0b Finnish	x01 Finland		x02 Mexico
x0c French	x01 France		
	x02 Belgium		
	x03 Canada		
	x04 Switzerland		
	x05 Luxembourg		
	x06* Monaco		
x3c Gaelic	x01* Scots		

	x03 Spain (Modern Sort)	x1d Swedish	x01 Sweden
	x04 Guatemala		x02 Finland
	x05 Costa Rica	x1e Thai	x01 Thailand
	x06 Panama		
	x07 Dominican Republic	x31 Tsonga	x01* Tsonga
	x08 Venezuela	x32 Tswana	x01* Tswana
	x09 Colombia		
	x0a Peru	x1f Turkish	x01 Turkey
	x0a Argentina	x22 Ukrainian	x01 Ukraine
	x0c Ecuador		
	x0d Chile	x20 Urdu	x01* Urdu
	x0e Uruguay	x33 Venda	x01* Venda
	x0f Paraguay		
	x10 Bolivia	x2a Vietnamese	x01 Vietnam
	x11 El Salvador	x34 Xhosa	x01* Xhosa
	x12 Honduras		
	x13 Nicaragua	x3d Yiddish	x01* Yiddish
	x14 Puerto Rico	x35 Zulu	x01* Zulu
x30 Sutu	x01* Sutu		
x41 Swahili	x01* Kenya		

Examples:

LANGUAGE x2b, x01	c Laotian standard
LANGUAGE x0d, x04	c Hebrew standard
LANGUAGE x200, x20	c User-defined language

Header Statement Reference

begin

```
begin > use(gnGroup)
```

The `begin` statement tells Keyman which group should be processed first when it receives a keystroke. This line originated in SIL-CC, and a simplified version was used in Keyman for consistency.

Example: `begin > use(main)`

BITMAP

```
BITMAP bmpFile
```

This statement replaces the `BITMAPS` statement from Keyman 3.x. Keyman 4 only requires one bitmap, to indicate that the keyboard is active. The bitmap is displayed at the bottom right of the screen, in the *tool tray*. This is a required statement.

COPYRIGHT

```
COPYRIGHT tsCopyrightMessage
```

Keyman 4 keyboards have a provision to display a copyright message when they are installed. This statement is optional.

HOTKEY header statement

```
HOTKEY tsHotKey
```

The `HOTKEY` statement specifies the hotkey that Keyman will use to turn the keyboard on. When this hotkey is pressed, any active keyboard will be turned off and the new keyboard will be turned on.

The hotkey can be any letter key, with any of the Shift, Control and/or Alt keys also held down. The specification of the `HOTKEY` statement follows the Microsoft standard for hotkeys in Windows. Inside a double-quoted string, you can combine the letter key with special characters to identify the shift state:

<u>To Combine With</u>	<u>Precede the letter-key by:</u>
Shift	+ (plus sign)
Ctrl	^ (caret sign)
Alt	% (percent sign)

Starting with version 3.1, the hotkey can also be in Virtual Key format, so that you can use any key on the keyboard.

```
HOTKEY "^+A"                    c Ctrl+Shift+A  
HOTKEY [Alt Shift K_PAUSE]    c Alt+Shift+Pause
```

LANGUAGE

```
LANGUAGE nLang, nSubLang
```

The `LANGUAGE` statement tells Keyman which language to associate the keyboard file with. See the **Language Codes** section for details on what the `nLang` and `nSubLang` parameters mean, and how to use them. The `LANGUAGE` statement is required.

LAYOUT

```
LAYOUT nID
```

The `LAYOUT` statement must be used when you are redefining a standard Windows keyboard layout. For instance, if you wanted to create an alternative English layout, you'd use `LAYOUT x5`. The majority of languages should be able to use layout `x1`, but if in doubt, just use a higher number.

English has four alternatives already, so you'll need start at x5; Portuguese, Italian, and German all has a single alternative layout, so for them start at x2.

MESSAGE

MESSAGE *tsMessage*

This is a generic message, such as a shareware notice that you can display when the keyboard is installed. This statement is optional.

Example: MESSAGE "This keyboard is freely redistributable."

NAME

NAME *tsKeyboardName*

The NAME statement lets you give a more descriptive name to your keyboard than just the file name. If NAME isn't specified in the keyboard file, Keyman will use the filename of the keyboard, excluding the extension, so the NAME statement is optional.

VERSION

VERSION *nKeyboardVersion*

The VERSION statement was added to Keyman 3.0 to allow later versions to easily distinguish what version of Keyman the keyboard was written for and handle it as such. The VERSION statement is required.

You must specify version 4 . 0 for Keyman 4.0 keyboards.

Body Statement Reference

any statement

`any(snStore)`

The any statement will, in effect, return true if the character input is in the store *snStore*. The character input is implied. This statement is only valid on the left side of a rule; the index statement is used to output the results of an any in the output. If an any is used in the key, it will be expanded out to include one rule for each character in the store. The any statement remembers the offset in the store where the match for later use with the index statement.

snStore: The name of the store to check in

```
+ any(keys) > index(output,1)
```

beep statement

`beep`

The beep statement produces a beep at the system speaker when the rule is matched. If you have a sound driver installed, beep will produce the sound specified by "Asterisk" in the Sounds option in Control Panel. When using the **beep** statement, *remember that it can delete all that was matched on the left side of the rule if you don't precede it with **context** or appropriate characters.* The **beep** statement is only valid in the output. The example given below will, if it receives a key that is in the *key* group, and the context ends with a *cons* character, ignore the *key* and leave the context alone.

no parameters

```
any(cons) + any(key) > context beep
```

context statement

`context`

The context statement simply reproduces the context stored from the rule match into the output. Use the context statement as much as possible as it is significantly faster than using the index statement.

no parameters

```
any(cons) "W" + any(key) > context index(keyout,3)
```

deadkey statement

`deadkey(nKey)`

The deadkey statement lets you program a deadkey in your keyboard. The deadkey will be the same as a normal character, but it won't come up on the screen. You can have up to 254 deadkeys, from 1 to 255.

nKey: A number from 1 to 255 that identifies the deadkey

```
+ '`' > deadkey(1)
deadkey(1) + 'e' > 'è'
```

group statement

`group(gnGroup) [using keys]`

group tells Keyman that a new group has started. There are two sorts of groups: key processing groups, and context processing groups. Key processing groups can include context checking, but context processing groups cannot include key checking. Keyman will use first the group specified in the begin statement, and move from there onto other groups. The keystroke received by Keyman is the same for all groups with key processing.

To tell Keyman that the group should include key processing, you should include the using keys section of the statement; if that is left out, Keyman assumes the group checks the context only. The keystroke will remain the same during processing; you can have many groups that each use using keys, and the keystroke will be the same for all of them. If you leave out the using keys bit, you have to also

leave out the '+' and the keystroke, because if you leave them in, the keystroke will be regarded as part of the context.

gnGroup: The name of the new group.

```
group(main) using keys
group(syllablecheck)
```

index statement

```
index(snStore, nOffset)
```

The index statement gets the offset of the character from the left side of the rule at offset *nOffset*. The offset refers to the position, including other characters, to the any statement which has saved the offset which it found the character in. The index will output the character at that offset from the store *snStore*. If used carefully, the index and any combination can be very powerful. The index statement is only valid in the output.

snStore: The store to output from

nOffset: The offset in the input to retrieve the any information from.

```
any(cons) "W" + any(key) > index(keyout, 3) "w" index(cons, 1)
```

match rule

```
match > esString
```

In each group, if Keyman finds a match rule, it will use it when a rule in the group was matched. A match rule can include use, return, beep and normal characters.

esString: The extended string to output, including the statements mentioned above.

```
match > use(AdjustVowels)
```

nomatch rule

```
nomatch > esString
```

nomatch is similar to match, but instead of the rule being used when a rule was matched, it will be used when a rule isn't matched in the group. A nomatch rule can include use, return, beep and normal characters.

esString: The extended string to output, including the statements mentioned above.

```
nomatch > beep
```

nul statement

```
nul
```

The nul statement will delete the context and key on the left hand side of the rule from the output; it is equivalent to having an empty output (which is not allowed). The nul statement probably will not be used often, because there are not many times you would want to delete the context and keystroke. The **nul** command must be the only character or command on the right hand side of the rule

no parameters

```
any(cons) + any(key) > nul    c delete consonant and next key
```

outs statement

```
outs(snStore)
```

The outs statement simply copies the store *snStore* into the position in which it has been inserted. Most of the time this is used only in stores but it can be used in the context and output as well.

snStore: The store to expand

```
store(key) "ABC" outs(DEFstore)
```


return statement

return will tell Keyman to stop processing rules and wait for the next keystroke to come. Keyman will not return to process other groups that called the one with the return statement.

no parameters

```
nomatch > return
```

store command

```
store(snStore) xsData
```

The *store* statement lets you store a string of characters or keys in a buffer which can then be referenced with any and index. Proper use of store can reduce many keyboards down to a few rules. A store is terminated at the end of the line (or continuation lines).

snStore: The name of the store to use

xsData: The data to place into the store *snStore*

```
store(keys) "ABCDEFGF"
```

use command

```
use(gnGroup)
```

The use statement tells Keyman to switch processing to a new group; after Keyman has gone through the new group, and any other nested groups, it will return to the previous one. The use statement can be used with the match and nomatch rules; it will work the same way.

gnGroup: The name of the group to switch control to.

```
any(Vowel) + any(DiacriticKey) > use(AdjustVowels)
```