Tavultesoft Keyboard Manager

Developer Documentation

VERSION 5.0

Tavultesoft

Last update: 07/08/00 9:05 PM

Later versions of this file may be available online at http://www.tavultesoft.com/keyman/docs/.

Chapter 1 Introduction....4
Documentation included in this distribution....4
Chapter 2 Writing a Simple Keyboard Program....5
Overview....5
Arranging the Layout on the Keyboard....5
Writing the Keyboard Program....6
Comments and Blank Lines....6
The Header....7
The Rules....8
Chapter 3 Further Programming....14
Unicode....14
Multiple Groups....14
The use Statement....14
The return Statement....15
The match and nomatch Rules....15
Summing Up....15
Constraints....15
Groups Without the “using keys” Keyword....16
Virtual Keys....16
Other Features....18
Other Header Statements....18
nul In the Context....18
The outs Statement....18
Long Rules....18
Chapter 4 Distributing Your Keyboards....19
Keyman File Types....19
Creating a Keyboard Package....19
Creating a Redistributable Installer....21
Technical Details....21

CHAPTER 1

Introduction

Welcome to the Tavultesoft Keyboard Manager. With the Tavultesoft Keyboard Manager (Keyman), it becomes practical to enter and edit documents that use languages and scripts other than English, for a wide variety of Windows application programs such as word processors, spreadsheets, databases, and desktop publishers.

Keyman has been developed with particular reference to the languages of South-East Asia and their scripts, but it can be readily adapted for many other languages. Keyman will allow you to mix many languages in one document, in your favorite word processor.

The most important feature of Keyman is the keyboard definition language that lets you develop your own keyboard layouts for just about any language.

This manual will guide you through the basics writing keyboards for Keyman, and explain the more advanced options that Keyman gives you. Reference information is given in the *Reference Documentation*.

Documentation included in this distribution

The following documents should be included with your Keyman distribution:

- **User documentation (Keyman50.pdf)**: Includes information on usage of Keyman 5.0. Does not include programming or development environment details. You should read this if you want to use Keyman, and it provides an overview of the functionality that developers will also find useful. This file will be included in redistributable versions of Keyman.
- **Developer documentation (kmdev50.pdf)**: This document. Keyboard developers should use this document to understand how to write keyboards and packages, and make the best use of the Keyman development tools. Also includes a tutorial on writing a simple keyboard.
- **Keyboard definition language reference (Km50lang.pdf)**: This file includes the details of the Keyman programming language (.kmn files), in a reference format.
- **Language code sheet (Langcode.pdf)**: A list of the MS-defined language codes. Developers should always check the Tavultesoft website for information on more recent updates to this document.
- **Keyboard template sheet (Keys.rtf)**: A useful template for documenting keyboard layouts.
- **Version information sheet (Version.txt)**: Contains information on changes, bug fixes and current version.
- **License information (License.txt)**: Contains details on the legal requirements for using Keyman, and licensing details.

Other documentation may become available and can be downloaded from http://www.tavultesoft.com/keyman/docs/.

CHAPTER 2

Writing a Simple Keyboard Program

There are two main steps to writing a keyboard program. The first step is to arrange the layout of the characters on the keyboard. Then, you enter the keyboard program according to your layout.

This chapter shows you the basic steps in writing a keyboard program; we will be using a simplified French keyboard as an example to follow through.

It is important to remember that a Keyman keyboard has a source file and a compiled file. The source file has the extension **.kmn**. The compiled file has the extension **.kmx**. You cannot installed a source file into Keyman 4.0, and you cannot edit a compiled file. See the section **Compiling Keyboards** for information on how to convert a source file into a compiled file.

Important Even if you have some experience in writing CC tables, you should still read this chapter, as it shows you the basic steps and structure of a keyboard file, which is slightly different to CC tables. However, your experience should help you learn the format more quickly.

Overview

You will be designing a simplified French keyboard for people who don't know the standard French keyboard layout; you will have to go through both steps mentioned above. This French keyboard (Quick French) doesn't follow the standard French layout; instead, it uses a basic English keyboard with some deadkeys to define vowel diacritics and other French characters needed.

You will need to know the character codes in the font that goes with this keyboard (for Quick French, Times New Roman and Arial work fine); the characters you will be using may be upper-ascii. If you do not have a font for the language you will be working with, you will need to obtain or create one; Keyman does not do anything about fonts.

Arranging the Layout on the Keyboard

First of all, you have to know what codes are used for the characters you are mapping with your keyboard program. (You can use the Character Map application that is provided with Windows to help you find these codes.)

For the Quick French keyboard, you will need all the vowels with different diacritics, some French symbols, and c-cedilla (upper and lower case); the codes needed are listed below, along with some others that are used by other European languages:

Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code
À	192	È	200	Ì	204	Ò	210	Ù	217		
à	224	è	232	ì	236	ò	242	ù	249		
Á	193	É	201	Í	205	Ó	211	Ú	218	Ý	221
á	225	é	233	í	237	ó	243	ú	250	ý	253
Â	194	Ê	202	Î	206	Ô	212	Û	219	Ç	199
â	226	ê	234	î	238	ô	244	û	251	ç	254
Ä	196	Ë	203	Ï	207	Ö	214	Ü	220	«	171
ä	228	ë	235	ï	239	ö	246	ü	252	»	187

Now that you know the character codes needed, you must decide how you want to key them. For the purposes of this example, the codes will be keyed according to the following table:

Code	Keys to create code
à, À, ...	backquote (`), followed by the corresponding vowel key
á, Á, ...	quote ('), followed by the corresponding vowel key
â, Â, ...	caret (^, SHIFT+6), followed by the corresponding vowel key
ä, Ä, ...	double quote ("), followed by the corresponding vowel key
ç, Ç	quote ('), followed by small or capital C.
«,»	double less-than symbol (<<) or double more-than symbol (>>)

The basic design of the keyboard is done. There will be more to come, but first you are going to write the first version of the keyboard program.

Writing the Keyboard Program

Before you start this section, start a text editor (such as Windows Notepad), and begin editing a new file. The Tavultesoft Integrated Keyboard Editor has been designed for this purpose and works closely with Keyman (see chapter 6). (DOS text editors are okay, too, but not quite as convenient.)

A keyboard program is divided into two sections: the header, and the rules. The header lets you define the name of the keyboard, hotkeys, and other general settings. The rules are divided into groups where you define how the keyboard responds to keystrokes. Every keyboard program must have both of these sections. A keyboard program normally has the extension .KMN and is an ANSI text file.

Note A keyboard program is in ANSI text format, not ASCII text format, because Windows uses the ANSI character set. If you are using a DOS text editor, you must remember that the characters you see on the screen aren't necessarily the same as the ones you'd see if you used a Windows text editor, such as Notepad.

Comments and Blank Lines

A Keyboard Program can have a comment at any point. The comment is identified by a letter 'c', with a space on either side (unless the comment is at the start of the line, in which case you only need a space on the right). The end of the line marks the end of the comment.

Blank lines are ignored by Keyman; you can have them anywhere in a keyboard program.

Examples:

```
c This is a comment

... c This is a comment, also
```

It is a good idea to use comments to document your keyboard program. They will help you remember why you did something in a particular way, and they will also help other people understand the program.

Add some comments to the start of the Quick French keyboard program, in your text editor, describing who wrote it, when it was written, and anything else needed, such as instructions on how to use the keyboard (from the tables above). You could even add a copyright notice, for a keyboard that uses complex algorithms, for example.

Example:

```
c
c Simplified French Keyboard Program for Keyman 5.0
c
c This keyboard program uses a simplified set of keys
c for typing French, especially for those who don't know
c the standard French keyboard.
c
```

```
c NOTE: This keyboard was created from the Keyman keyboard
c programming tutorial.
c
c Written by Anybody, August, 2000
c
```

The Header

The header is easy to create; it consists of statements that help Keyman identify the keyboard and set default options for it. Each statement in the header must be on a separate line and is usually written with capital letters, although that is not required. Keeping the statements in the header in upper case helps you identify them easily, and keeps them consistent with keyboard programs other people might write.

Five statements are required in the header. Some other optional statements are described in later chapters. The required statements are: `NAME`, `BITMAP`, `LANGUAGE`, `VERSION`, and `begin`. The `begin` statement is usually written in lower case.

The NAME Statement

The `NAME` statement tells Keyman the long descriptive name of the keyboard, which can be as long as eighty characters. This name is used in the Keyman menu, and in the Options dialog box. The name must be enclosed in double quotes ("). Any character except the double quote is legal within the name.

For our Quick French keyboard, we will use the name “Quick French.” Add the `NAME` statement to the keyboard program, as follows:

```
NAME "Quick French"
```

You can give your keyboard program a different name, if you wish.

The BITMAP Statement

The `BITMAP` statement tells Keyman which bitmap file is used for the keyboard icon. The bitmap uses the standard Windows .BMP format; you can create them using Paintbrush. You should make it 16x16 pixels, as that will be the final display size. The bitmap can be monochrome or color, but the final .BMP file must be less than 64 kilobytes. The `BITMAP` statement accepts the name of the bitmap file, in upper or lower case, with or without the .BMP extension included. The file name is **not** enclosed in quotes.

Add the `BITMAP` statement, as follows:

```
BITMAP FrKey
```

The VERSION Statement

The `VERSION` statement is the simplest statement; for a keyboard intended for version 5.0 of Keyman, simply add **5.0** to the end of the line.

The Quick French keyboard is intended to be used in Keyman 4.0. Therefore, add the following line to your program:

```
VERSION 5.0
```

The LANGUAGE and LAYOUT Statements

Conversely, the `LANGUAGE` statement is the most difficult header statement to understand. For the purposes of this example, we will just give you the complete statement. You will need a `LAYOUT` statement as well, as this is an alternative to the standard French keyboard. These statements are described fully in the Language Reference.

```
LANGUAGE x0c, x01           c French (France)
LAYOUT x01
```

The begin Statement

The `begin` statement tells Keyman which group to start processing with when it receives a keystroke. Later on you will learn how to use multiple groups to process keystrokes, but at present all you need to know is to include this line in the header.

For the Quick French keyboard, add the following line to tell Keyman to start in the `Main` group:

```
begin > use(Main)
```

Conclusion

Those five statements are the only ones required in the header. You can add comments to the ends of the statements to help other people understand them.

Your Quick French keyboard should so far look like this:

```
c
c Simplified French Keyboard Program for Keyman 5.0
c
c This keyboard program uses a simplified set of keys
c for typing French, especially for those who don't know the
c standard French keyboard.
c
c NOTE: This keyboard was created from the Keyman keyboard
c programming tutorial.
c
c Written by Anybody, August, 2000
c

NAME "Quick French"
BITMAP FrKey
LANGUAGE x0c, x01
LAYOUT x01
VERSION 5.0

begin > use(Main)
```

The Rules

Before we start on the rules, we will define some terms:

Term	Definition
rule	A rule tells Keyman what output is associated with a keystroke under certain conditions; it is divided into three parts: the context, key, and output
context	The context specifies the conditions under which the rule is acted upon. It is compared with the most recent characters output.
key	The key is the code for a single keystroke that the rule acts on.
output	The output is the part of the rule that defines what characters are to be put on the screen when the rule's conditions are met.

The Groups

The first thing to do is to define the group. There are two types of groups: one that processes the key pressed and the context and another that does further processing using only the context. For most purposes, the first type of group will do all you need. The group of rules ends at the next `group` statement or at the end of the file if there are no more `group` statements. We said in the section about the header that the group to be processed first would be identified by the `begin` statement.

The `begin` statment defined the first group as `Main`. In your program, add a new line:

```
group(Main) using keys
```

`using keys` tells Keyman that the group processes keystrokes as well as context. If you leave this out, the keystrokes will be ignored.

Simple Rules

The simplest rule you can have tells Keyman to convert one key into another. A rule of this sort is represented in Keyman in the following way:

```
+ key > output
```

where *`key`* is the key to be translated, and *`output`* is the character to be output. The plus sign (+), is required, and shows you that the next character in the string is the keystroke. **Note:** *This has changed from Keyman 3.x, where the plus sign was optional.* More complex rules can have characters before the plus sign (the context). The right angle-bracket (>) tells Keyman which part of the rule is output and which part is the key and context. Single characters can be represented in several different ways; the possible methods are listed below:

Representation	Example of the character "x"
Inside single quotes	'x'
Inside double quotes	"x"
As a decimal number (base-10)	d120
As a hexadecimal number (base-16)	x78
As an octal number (base-8)	170

These different ways of representing a single character follow the SIL-CC conventions. The first three ways are the most often used, and octal representation is rarely used. Multiple characters (a *string*) can be represented in quotes simply by having more than one character in the string. You can have any combination of these representations in a rule, with spaces between them.

For example, to convert the key "a" to the character "z", you would include the following line in your keyboard program:

```
+ 'a' > 'z'
```

Or, to convert "?" to "Hello World!", you would have this line:

```
+ "?" > "Hello World!"
```

You can use either single or double quotes.

A use of the decimal representation is, for example, in the British English keyboard, where the hash sign (#) is converted into a pounds sign (£, decimal code 163):

```
+ '#' > d163
```

Using the Context in More Complex Rules

Often you will want to know the previous characters that have been typed and translate the keystrokes accordingly. *Keyman remembers the characters that came out on the screen, and not the actual keys typed.* It is important to remember this, because some programs such as SIL's Keyswap (for DOS) work with sequences of keys rather than characters. The characters that came out on the screen are called the *context*. The context is represented in a rule to the left of the keystroke, before the plus sign. For example,

```
"^" + "e" > "ê"
```

In this example, if you type a "^" (caret) followed by the letter "e", it will come out with the European letter "ê". The caret is the context, the letter "e" is the key, and the letter "ê" is the output.

You can add some of the rules to the Quick French keyboard program now. Add the rules for all the "a"-related characters; you will quickly see how many rules it would require for a complex keyboard. Another example rule for the Quick French keyboard program is:

```
'`' + 'a' > 'à'
```

The Any, Index, and Store Statements

Keyman lets you translate a group of characters in one rule. It does this with the `any`, `index`, and `store` statements. A `store` statement creates a set of characters that can be operated on together under a name. The `any` statement lets you match a character in a store and the `index` statement lets you output a selected character from it.

A `store` statement comes between the `begin` statement and the first group. It must all be on one line. (No `endstore` statement is required as in SIL-CC; in fact, it is not supported.) The statement has the following syntax:

```
store(name) string
```

name is the name to give to the store. A store name can be up to 16 letters long, but it is usually best to keep it short. The name can be any combination of letters and numbers; spaces and punctuation characters are illegal. The second part of the statement, *string*, is the string to put in the store; it can use any combination of the character representations talked about in the previous sections. An example:

```
store(lwrvowel) 'aeiou'
```

In this example, Keyman will create a store called "lwrvowel", and make the contents of the store equal to "aeiou".

To use a store, you must have an `any` statement on the left hand side of a rule, and, optionally, a corresponding `index` statement on the right hand side.

An `any` statement allows you to designate a set of characters instead of a single character for the key or, as part of the context. The syntax of the statement is:

```
any(storename)
```

For example, you could have the following:

```
store(stops) '!.?'
.
.
.
+ any(stops) > 'GO!'
```

This example would convert any of the characters "!", ".", and "?" to a "GO!". (This example is actually not very useful.)

But, to make the `any` statement useful, you really need to have a statement that lets you know the matched character. The `index` statement lets you do that.

The `index` statement lets you output a character in a store that is at the same position as the matched character from the equivalent `any` statement's store. The `index` statement has the following syntax:

```
index(storename,offset)
```

The *storename* is obvious; however, the *offset* part needs some explaining. As Keyman allows you to have more than one `any` statement in a single rule, the `index` statements in that rule need to know which `any` statement they are to take their matched character information from. The *offset* parameter tells Keyman the position of the character of the `any` statement that is to be used, with the first character of the context having the offset 1. For example,

```
+ any(lwrvowel) > index(uprvowel,1)
```

This rule would convert all lower case vowels to upper case. Or,

```
any(stops) + any(lwrvowel) > index(stops,1) index(uprvowel,2)
```

This one capitalizes any lower-case vowel following a full-stop, question, or exclamation mark.

The context Statement

If the context of the rule is not modified in the output, then you can replace the `index` statements on the RHS of the rule with a `context` statement. For example, the previous rule becomes:

```
any(stops) + any(lwrvowel) > context index(uprvowel,2)
```

This is faster and, for more complex rules, easier to read. Use the `context` statement wherever possible in preference to using `index` statements.

The Quick French example keyboard can make use of this quite easily; an example will be shown for "^" and a vowel:

```
store(vowel) 'aeiouAEIOU'
store(caret) 'âêîôûÂÊÎÔÛ'
.
.
.
'^' + any(vowel) > index(caret,2)
```

You should be able to add all the rest of the rules fairly easily. At present, leave out the "«", "»", and c-cedilla rules. For the "ý" and "Ý", just add a single rule (don't use `any` and `index`). You can now delete the single rules applying to "a".

So far, your Quick French keyboard should look like this:

```
c
c Simplified French Keyboard Program for Keyman 5.0
c
c This keyboard program uses a simplified set of keys
c for typing French, especially for those who don't know the
c standard French keyboard.
c
c NOTE: This keyboard was created from the Keyman keyboard
c programming tutorial.
c
c Written by Nobody, August 2000
c

NAME "Quick French"
BITMAP FrKey
LANGUAGE x0c, x01
LAYOUT x01
VERSION 5.0

begin > use(Main)

store(vowel) 'aeiouAEIOU'
store(caret) 'âêîôûÂÊÎÔÛ'
store(acute) 'áéíóúÁÉÍÓÚ'
store(grave) 'àèìòùÀÈÌÒÙ'
store(colon) 'äëïöüÄËÏÖÜ'

group(main) using keys

"'" + 'y' > 'ý'
"'" + 'Y' > 'Ý'
"^" + any(vowel) > index(caret,2)
"'" + any(vowel) > index(acute,2)
"`" + any(vowel) > index(grave,2)
'"' + any(vowel) > index(colon,2)
```

```
c End of file
```

Testing the Keyboard

Before you go any further, you should test your keyboard. Save your file, and start a command prompt. You can use TIKE to compile your keyboard, or type the following command from the Keyman directory to compile your keyboard:

```
KMCOMP QFrench.kmn QFrench.kmx
```

If any errors occur, refer to the section **Compiling Keyboards**.

After this, install the keyboard using the Windows Explorer. You should be able to use the keyboard in a word processor now.

Load a text-editor or word-processor, such as Notepad, or WinWord. Select the Quick French keyboard and try it out. Type sequences like **^a^e'a'e**. Once you are sure that that is all right, then try typing something like this: **"A problem in the keyboard."** (Include the quotes.) You can see the problem: when you type something in quotes, if the letter after the quote character is a vowel, it will be converted.

Fixing the Problems

Open up your keyboard program again. The problem exists with two lines; both of the lines regarding quotes will need to be changed. But first you have to decide how you are going to represent the quote character when it is to be used as a quote character. Probably the easiest way is just to type it twice.

The line you need to add is this; this will fix it for double quotes; add another line to fix it for single quotes.

```
'"' + any(vowel) > '"' index(vowel, 3)
```

Another thing that would be nice is to make the diacritics as *deadkeys*. A deadkey is a key that does not come out on the screen when it is pressed, but is still remembered in the context. Many European keyboards use deadkeys.

We will show you the line needed; you will need to remove the old rule to do with carets and explain it for the caret (^) character:

```
+ '^' > deadkey(1)
deadkey(1) + any(vowel) > index(caret,2)
```

The `deadkey`, or `dk` statement accepts a number identifying it; it will not appear on the screen, but it does stay in the context. You can have up to 254 different deadkeys, starting from 1.

You will want to add the deadkey rules for all the other characters; don't forget to use a different deadkey identifying number for each one. You will also need to modify the quote modification statements talked about above, to work with the deadkey better; it becomes simpler, as shown below:

```
dk(2) + '"' > '"'
```

You should add this sort of statement for all the diacritics, in case you wish to use the original character.

There are some other characters we haven't got support for yet: «, », ç, and Ç. We decided to represent the "«" and "»" characters with double less-than and double more-than symbols. You should be able to add rules for these, as well as for the "ç" and "Ç" symbols.

But what if someone wanted to type "<<<<<<< *** >>>>>>>", for instance, as a divider to a section of a book. They wouldn't want it to come out as "«««< *** »»»>". So we will make use of a deadkey to have it come out correctly, as shown below:

```
"«" + "<" > "<<" dk(5)
"»" + ">" > ">>" dk(5)
dk(5) + "<" > "<" dk(5)
dk(5) + ">" > ">" dk(5)
```

You should be able to see what this does. Test your keyboard again; there should not be any more problems. You have completed the Keyman keyboard tutorial.

The following chapter will explain some of the more advanced features of the Keyman keyboard language; you could use them to extend this keyboard.

CHAPTER 3

Further Programming

This chapter will build on what you have already learnt from chapter 4; it will be assumed that you understand the basic Keyman keyboard program format.

The following subjects will be discussed in this chapter:

- Unicode
- Multiple groups
- Constraints
- Groups without the "using keys" keywords
- Virtual keys
- Other features

Unicode

Keyman 5.0 adds support for Unicode. There are two changes to the keyboard file format to support Unicode: the `begin` statement, and the addition of the `U+`***`xxxx`*** character descriptor. Keyboard files can contain either or both Unicode and ANSI rules; Keyman will determine whether the application in use supports Unicode and choose which begin group to use accordingly.

The `begin` statement has the following options:

```
begin Unicode > use(myUnicodeGroup)
begin ANSI  > use(myANSIGroup)
```

The `ANSI` text is optional; this provides seamless backward compatilibity with version 4.0 of Keyman.

Inside a group, the context and output of a rule support Unicode characters. You cannot use Unicode characters to specify a key – this is not a limitation, as you can specify all keys with ANSI characters or virtual key combinations anyway. Unicode characters must be specified with a `U+`***`xxxx`*** description at present – no support is available for Unicode format keyboard source files.

For example, in Lao,

```
U+0EAB + 'o' > U+0EDC
```

will produce the following output:

context is
press **o**, which corresponds to
output is the combined character

Multiple Groups

In chapter 4, you learnt how to create a file with one group. Multiple groups can be useful for doing further processing such as changing characters in certain contexts, or, as is done for some South-East Asian languages, syllable splitting.

The use Statement

A group can be added with the `group` statement; the previous group ends at the line before the statement. However, to use the group, you must have a way of jumping from one group to another. The `use` statement lets you do this. This statement is legal on the right hand side of a rule; you can put it anywhere in the string, with one limitation: no `index` or `context` statements are allowed after the `use` statement; using them will cause run-time errors.

Any output in a group will affect groups called by it, as well as groups after it. The current context will be modified to add the changes made by a group, before returning to the previous group, or jumping to another one.

The `use` statement has the following syntax:

```
use(groupname)
```

Where *`groupname`* is the name of the group to jump to. After the new group has finished processing, control will return to the statement after the current one, in the same rule. You can nest quite a few groups; the exact number is not known.

The return Statement

The `return` statement stops all processing of rules and returns control to the typist. No statements are executed after the `return` statement, even if it jumps back through multiple groups.

The `return` statement has no parameters; it must be on the right hand side of the rule.

The match and nomatch Rules

Two special rules can be included in a group: the `nomatch` rule and the `match` rule. One of these two rules will be executed every time the group is entered, unless the rule matched contains a `return` statement.

The rules are represented in the following way:

```
nomatch > right-hand-side
match > right-hand-side
```

Where the *`right-hand-side`* can include any of the statements legal to the right hand side of a rule, except for `context` and `index`. Both of these rules can jump to other groups with the `use` statement, output characters, or stop processing with the `return` statement.

If no rule is matched, the `nomatch` rule will be executed; if a rule is matched, the `match` rule will be executed, *after* the matched rule has been executed.

Summing Up

Several of the example keyboards on the Tavultesoft website illustrate the usage of these statements and rules. To sum up:

```
LHS > RHS or use(group) or return
match > RHS or use(group) or return
nomatch > RHS or use(group) or return
```

Constraints

Constraints are ordinary rules that restrict certain combinations from being typed. These rules can occur anywhere, even in a `nomatch` or `match` rule.

A constraint rule can just be something like the following line:

```
any(vowel) + any(vowel) > context
```

This would restrict two vowels from being typed in a row; the second key would just be ignored.

However, you might wish to let the typist know that they typed an illegal combination. This can be done with the `beep` statement. The `beep` statement simply makes a beep at the PC speaker.

Note If you have a sound driver installed, such as the PC Speaker sound driver, or a driver for a sound card, the `beep` statement plays the sound identified by the Asterisk entry in the Sounds option in Control Panel.

The `beep` statement is legal only on the right hand side of a rule; it just tells Keyman to make a beep, nothing else. A `beep` statement can be used both in `match` and `nomatch` rules. Examples of the beep command:

```
any(vowel) + any(vowel) > context beep
+ any(illegal) > beep
```

When you are restricting a set of keys, without context, from being typed, but you don't want a beep, another statement is required. The `nul` statement tells Keyman that nothing is on the right hand side of the rule. The rule above would become:

```
+ any(illegal) > nul
```

This would simply ignore any illegal keys. In some situations with multiple groups, this can be more useful than it appears. The `nul` statement is not necessary for the `nomatch` and `match` rules; just don't add them if you don't want them to do anything.

Typically, you would put constraints in the first group of a keyboard program, and every rule matched would simply be a rule testing for illegal context and key combinations. The `nomatch` rule would then be:

```
nomatch > use(maingroup)
```

Obviously, you could name the next group anything you like. The `match` rule would probably be left out.

Groups Without the "using keys" Keyword

The `using keys` keyword, introduced in chapter 3, was used in a `group` statement to tell Keyman that the group would be needing information on the key pressed. In some situations, you might want ignore the keystrokes sent, such as for syllable splitting, or for changing the order of stacked diacritics, which only depends on the context.

Virtual Keys

With what you have learnt so far, any letter, number, or punctuation mark can be identified as a key in a rule. However, you cannot test the Ctrl and Alt states of these keys; with some keyboards, it is necessary to do so.

Virtual keys allow you to do that. A virtual key keyword can identify almost any key on the keyboard; a few specialized ones are either reserved or unable to be used.

Virtual keys are allowed only in the key section of a rule, not in the context or the output. A virtual key is identified by an opening bracket character ('['). It ends at a closing bracket character (']'). Inside the brackets, you can have a combination of shift-key codes and the actual virtual key, which is identified by a "K_" at the start of the keyword.

The keyboard shown further on gives the virtual keys for all keys on the standard US 101 key keyboard. (**Note**: The arrangement may not be identical to your keyboard.)

Important You must not use virtual keys in the output. Keyman will recognize them, but output will be garbled. This version of Keyman does **not** support virtual keys in the output.

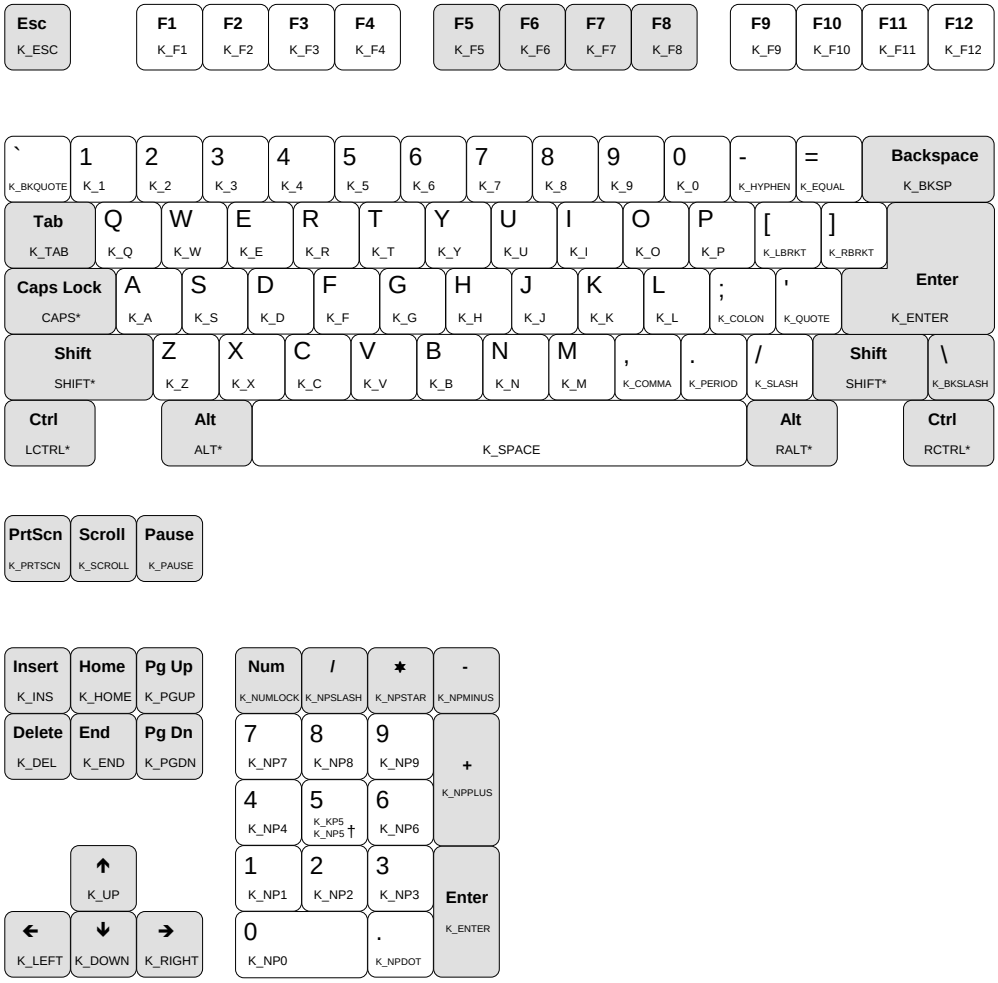


Figure 5.1: The Virtual keys keyboard layout

* Keys marked by a star are special keys that will be discussed in more detail further on.

† This key can be either `K_KP5` when NUM LOCK is off, or `K_NP5` when NUM LOCK is on; this applies to all the keys on the number pad - when NUM LOCK is off, the movement keys will be used.

For example, to test for the SCROLL LOCK key, you would have the following line:

```
+ [K_SCROLL] > output
```

If you want to test for a key that is used with SHIFT, CTRL, ALT, or CAPS LOCK, then you would proceed it with one of the following keywords:

Shift key to test	Keyword
SHIFT	`SHIFT`
Either CTRL	`CTRL`
Left CTRL	`LCTRL`
Right CTRL	`RCTRL`
Either ALT	`ALT`
Left ALT	`LALT`
Right ALT	`RALT`
CAPS LOCK on	`CAPS`
CAPS LOCK off	`NCAPS`

So, if you wanted to test for Right ALT + the letter “e”, you would have the following line:

```
+ [RALT K_E] > output
```

This version of Keyman does not let you use stores for virtual keys.

Other Features

Other Header Statements

There are three optional header statements that Keyman recognizes, all working with CAPS LOCK.

The first statement, `CAPS ALWAYS OFF`, makes sure that CAPS LOCK cannot be turned on while the keyboard is active, and it turns CAPS LOCK off when the keyboard is switched on. Put this statement on a single line in the header, as follows:

```
CAPS ALWAYS OFF
```

The other two statements, `CAPS ON ONLY`, and `SHIFT FREES CAPS` are usually used together. `CAPS ON ONLY` makes the CAPS LOCK key like a typewriter CAPS LOCK, where pressing it turns it on only. `SHIFT FREES CAPS` tells Keyman to recognize SHIFT and turns capitals off. Using these two together makes Keyman work like many European keyboards. These two statements each take a single line in the header, as shown below:

```
SHIFT FREES CAPS
CAPS ON ONLY
```

nul In the Context

The `nul` statement is used at the start of the context to tell Keyman only to match that rule if there are only as many characters output on the screen as in the context. This statement is not very likely to be used; there is a possibility you may use it for testing after a keyboard has been turned on, or to change a character into the best possible output without knowing what is before it. For example,

```
nul + 'a' > 'A'             c Not very useful!
```

The outs Statement

The `outs` statement places the content of a store into the string at its position. You would probably only use the `outs` statement for creating large stores. Usage:

```
outs(storename)
```

Long Rules

When you are making your keyboard, you may find that some lines are very long and are hard to read if made shorter. Keyman has a way of getting around this: by putting a backslash (\) on the *very end* of the line, Keyman is told that the line should be joined with the next one. You can do this for multiple lines if necessary, up to 1K (about 1000 characters) long. The backslash must come *after* comments if you have them. For example,

```
any(LowerCaseVowel) + any(UpperCaseVowel) > \
    index(UpperCaseVowel,1)       c From previous line  \
    index(LowerCaseVowel,2)       c From previous line
```

CHAPTER 4

Distributing Your Keyboards

Keyman 5.0 provides several options for distributing your keyboards. These are listed in order of complexity, and each distribution method builds on the previous method. Keyman 4.0 only supported the first method.

Keyboard file: This is your basic compiled keyboard file. Simple to create and simple to use — just double click to install. Uninstallation is provided through the Add/Remove Programs Control Panel applet. The user must have Keyman installed to use a keyboard file.

Keyboard package: This contains a keyboard file, plus fonts, and any additional files you wish to include. The end user can install this by double clicking, and installation is as simple as with the standard keyboard file. Uninstallation is similar to a keyboard file. The user must have Keyman installed to use a keyboard package.

Redistributable installer: This contains one or more keyboard packages, plus fonts and additional files, and the Keyman redistributable files. This is a totally standalone package, and the user can install it by double clicking and following the instructions on screen. Users can choose to install any or none of the keyboards packages included in the installer. Uninstallation is again available through Control Panel.

Keyman File Types

Keyman has the following file types:

A keyboard source file (**.kmn**). This file contains the instructions which the Keyman compiler will use to create a keyboard file that can be used with Keyman. You need TIKE to use this file.

A compiled keyboard file (**.kmx**). You can install this into Keyman. This is the primary file type in Keyman.

A keyboard package source file (**.kps**). This file contains a list of files and other details that will be packaged up by the Keyman compiler to create a keyboard package. You need TIKE to use this file.

A keyboard package file (**.kmp**). This file contains a keyboard, plus fonts, instructions for use and any other files that a keyboard developer wants to include.

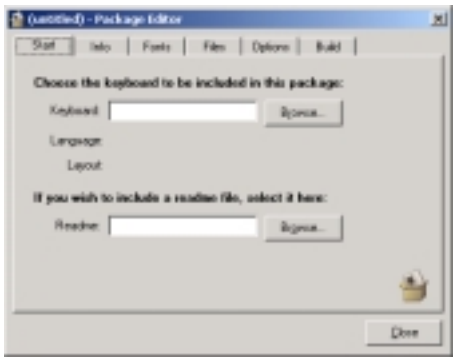
A redistributable installer (**.exe**). This file contains one or more keyboard package files, plus the necessary files to run Keyman on an end-user system. It does not include the Keyman development environment or other files.

These files types are

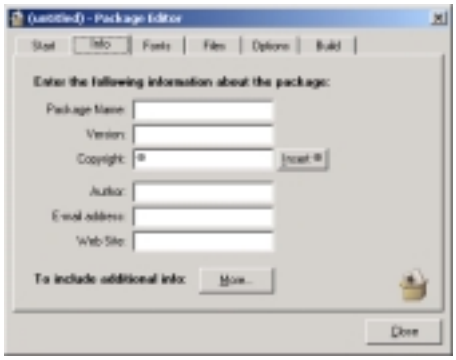
Creating a Keyboard Package

Once you have completed writing your keyboard, you will probably want to package it with fonts and instructions on how to use it. In TIKE, you can create a package from the File/New menu option.

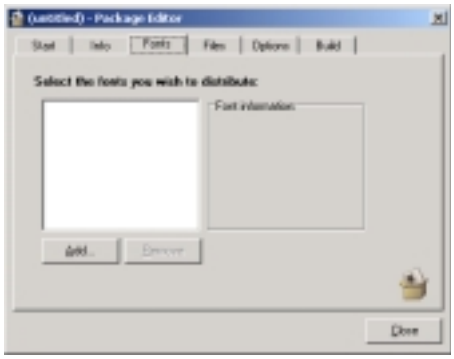
The following dialog will appear. You should click the tabs at the top to fill in details for each page of the dialog.



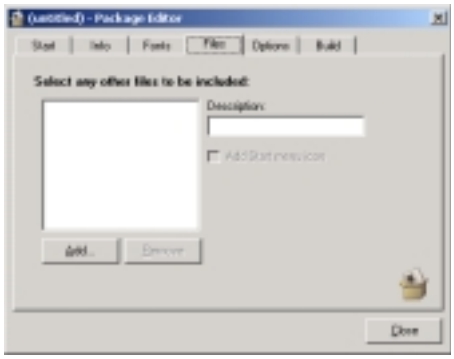
You should fill in details of your keyboard -- name, version, copyright, and contact details here. These details will appear when you view a keyboard package or install a keyboard package. If a Start Menu folder is created, it will have the name of the package as given here.



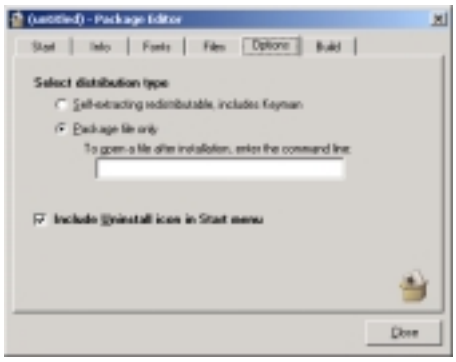
You can add fonts in the following page -- this is pretty self explanatory. If the font is already installed, Keyman won't install it again.



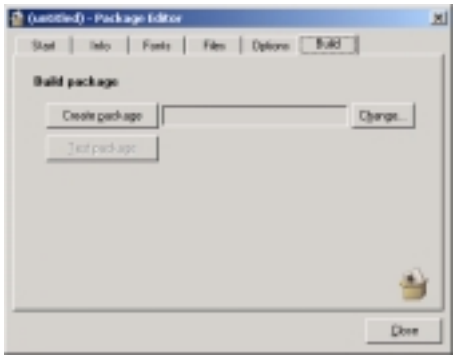
Any additional files that you want to include can be listed here. You can also list these files in the Start Menu, which can be useful for documentation, etc.



You can create either a self-extracting executable, identical to the redistributable installer, or just a package. If you want multiple keyboards in one distributable file, you should create a package file only. If you intend to distribute the keyboard in a package, you should probably not include an Uninstall icon in the Start Menu.



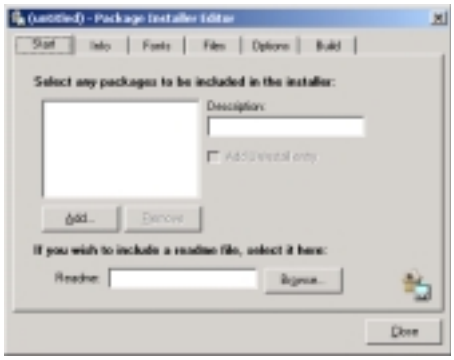
Once all the details are filled in, you must save the package before attempting to build it. You can build the package with the "Create package" button, and then once built, you can test it.



Creating a Redistributable Installer

Creating a redistributable installer is very similar to creating a keyboard package. You create a new installer from the File New menu option.

The only page which is significantly different is the first page, which lets you select the packages you want to install together.



You should fill in all appropriate details on the subsequent pages, and then save the package source. Clicking Create Package will then create the installer. This should be tested to ensure that you have correctly configured your package.

Technical Details

Keyman packages use the industry standard ZIP file format. You can open them using WinZip or other archiving utilities. Keyman provides a facility for this in the the context menu for the package (available by clicking right mouse button on the icon in Explorer). A special file, kmp.inf, included in the archive, contains all the details that Keyman uses to control installation of the keyboard.
Information on the kmp.inf and kmredist.inf file formats will be included at a later date.