4 Physical Keyboard Programming

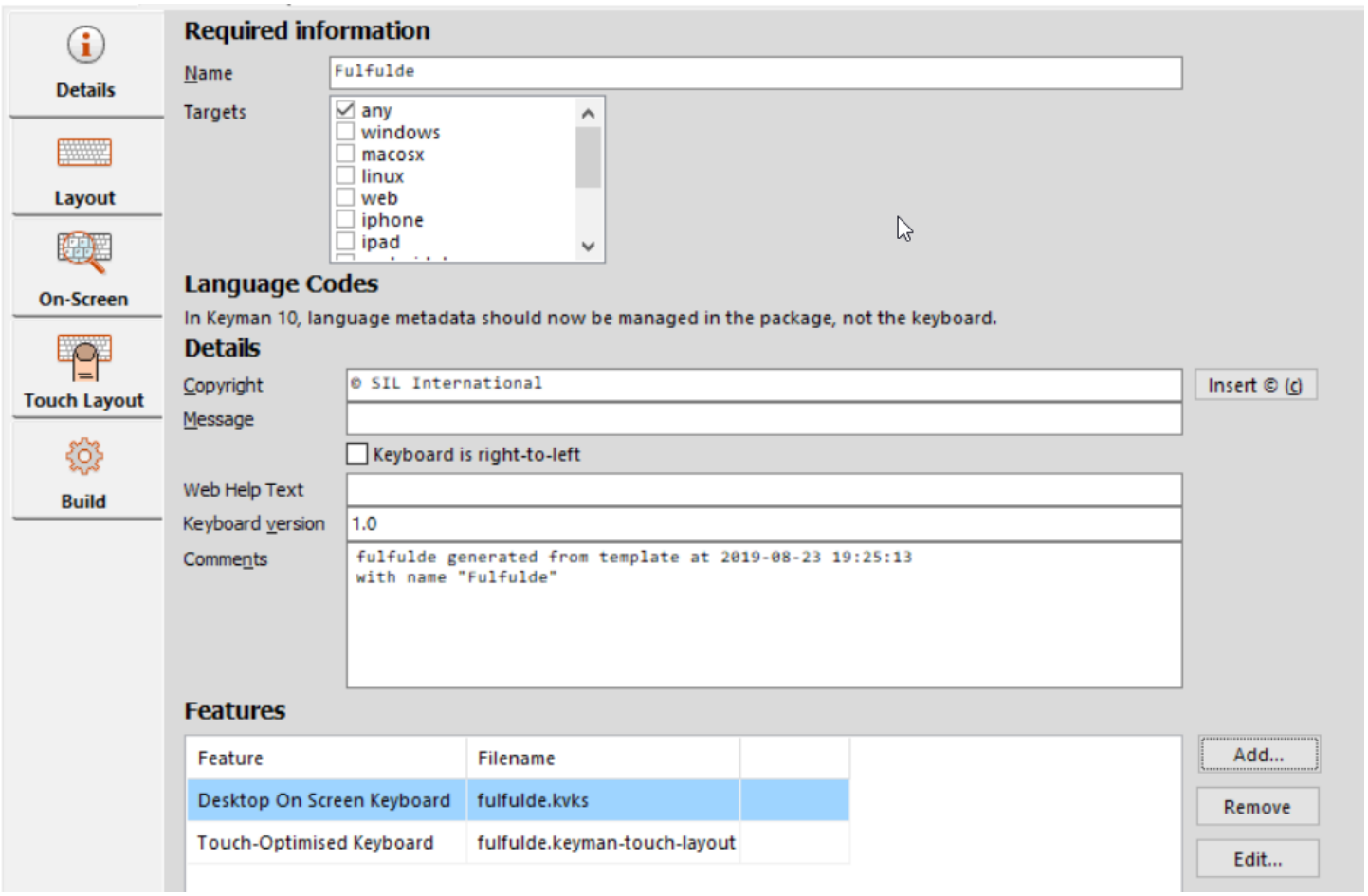
If you have planned your keyboard using the physical keyboard design template, there are some keys that you were able to swap with existing keys with no additional coding required on the **Design** tab. But you probably won't be able to do everything visually. If your keyboard uses dead keys, or rotas, or combinations to modify output, then you will need to do some coding on the **Code** tab.

Coding your keyboard

First, open the keyboard file that was created when you created your project. For this example, we will use the Fulfulde keyboard.

On the keyboard tab of your new project page, Keyman has already created the fulfulde.kmn file. Click on it to open the file for editing.

The editor page should look like this:



Details tab

Two features were added by default: the desktop on-screen keyboard, and the touch-optimized keyboard. These show up as additional tabs on the left. If you don't see a tab for On Screen and Touch Layout on the left, then you will need to click the Add button above and add them one at a time.

If you want to add a small icon to identify the keyboard, Keyman will use it in place of its generic icon. Click on the "Add..." button and then select "Icon". Keyman Developer includes a way to edit the icon, or you can edit the icon with another program. For this tutorial, we will use the generic icon.

Working with an existing keyboard? If you are adding touch capability to an existing keyboard, it is good to follow the project creation and then add your existing .kmn file to the project. You will need to add the two features above and to copy the header information from the auto-generated keyboard file to the beginning of your existing file. (Or, alternatively, copy information from your existing .kmn file into the newly created one.)

Layout (for typing keyboard)

On the bottom of the Layout tab, there is a Code tab. This is where you add and modify the code. For Fulfulde, we need to produce the following characters:

□□□□□□□

ɓɗŋɲƴƁƊƖŋƝƳ

This is a simple case to start with. There are only 5 special characters, lowercase and uppercase.

We will use the semi-colon as the combining key to create a number of these. A semicolon is usually followed by a space, so if we type a semicolon, followed by another character, we can output whatever we want.

```
;b > ɓ
```

We will use the semicolon for four of these, but we have to do something different for the ɲ since we can't use semicolon n to produce both the ŋ and the ɲ. So for the ɲ we will use j + n. We can do this because n never follows a j in the language, and ɲ looks like jn stuck together.

Using the simple method below, we can do this in 10 rules following the keyman rule structure **Context + keystroke > output**

Simple


```
begin Unicode &gt; use(main) group(main) using keys &quot;;&quot; + &quot;b&quot; &gt; &quot;ɓ&quot; &quot;;&quot; + &quot;d&quot; &gt; &quot;ɗ&quot; &quot;;&quot; + &quot;n&quot; &gt; &quot;ŋ&quot; &quot;;&quot; + &quot;y&quot; &gt; &quot;y&quot; &quot;;&quot; + &quot;B&quot; &gt; &quot;Ɓ&quot; &quot;;&quot; + &quot;D&quot; &gt; &quot;Ɗ&quot; &quot;;&quot; + &quot;N&quot; &gt; &quot;Ŋ&quot; &quot;;&quot; + &quot;Y&quot; &gt; &quot;Y&quot; &quot;j&quot; + &quot;n&quot; &gt; &quot;ɲ&quot; &quot;j&quot; + &quot;N&quot; &gt; &quot;Ɲ&quot;
```


This works, and you could stop here. But there is a more efficient way to write this code, which we will now try in the advanced section below.

Advanced (using only three rules)

In Keyman, you can create variables that hold a group of characters. We will create two groups of equal size, and then in the rule, we will tell Keyman to replace a member of the one group with the corresponding member of the other group.

See how the base characters in the store are in the same order as the output characters in the output store. (The store names can be anything that you want.)


```
begin Unicode &gt; use(main) store(basekey) &quot;bdnyBDNY&quot; store(output_char) &quot;ɓdŋyƁƊƝY&quot; group(main) using keys &quot;;&quot; + any(basekey) &gt; index(output_char,2) &quot;j&quot; + &quot;n&quot; &gt; &quot;ɲ&quot; &quot;j&quot; + &quot;N&quot; &gt; &quot;Ɲ&quot;
```


With the stores in place, it takes only **one rule** to handle most of the special characters!

Let's look at that rule:

';' + any(basekey) > index(output_char,2)

This says, "When there is a semicolon in the context and one of the characters in basekey is typed, output the character at the same index point **for the second argument** on the left side of the expression."

The number 2 in the index is known as the offset. It is not referring to the second character in the index. Instead, it is saying that the index is based on the second argument on the left.

This example will help clarify. If we wanted to produce a special character every time the base character was typed (swapping a ɓ for every b), then we would remove the semicolon context and write the rule this way:

+ any(basekey) > index(output_char)

Now the index defaults to the first argument on the left side of the >. You don't need to include the offset at the end of the index since 1 is the default.

Try it

You can copy and paste the simple or advanced code above into your Fulfulde keyboard example. Just make sure you don't modify the header statement that is already there!

Test it

Once the code has been added, you can test the keyboard.

1. Click the Save button
2. Select Keyboard -> Include Debug Information
3. Compile the keyboard. F7 or Keyboard -> Compile keyboard
   If it compiles well, you will see
   - Keyboard Ideas and Samples
4. Select Keyboard -> Test keyboard and a window will open allowing you to type and test the character sequences.
   You may need to right-click in the window to set the font to be used.

Keyman Developer has a powerful debugger tool that can help you troubleshoot when your rules are not working as expected. Unfortunately, we don't have materials at this time to explain the use of the debugger, but it works similarly to other compiling environments.

Using stores to do most of the work

You can make two stores the same length, one with the base character and the second with the alternate character, like this:

```
store (baseChar) &quot;aA&aacute;&Aacute;cCdDeEgGhHiInNoO&oacute;&Oacute;pPqQ&oslash;&Oslash;sStTuUyY&quot; store (altChar) &quot;&aacute;&Aacute;&auml;&Auml;&ccedil;&Ccedil;đĐ&eacute;&Eacute;ġĠħĦ&iacute;&Iacute;&ntilde;&Ntilde;&oacute;&Oacute;ǒǑṗṖ&oslash;&Oslash;ǿǾŝŜŧŦ&uacute;&Uacute;ỳỲ&quot;
```

Each character in the store must be a single codepoint! All of the accented characters in this store are Composed Characters, that include the accent in the glyph. Because stores count codepoints and not visible characters, if you include an accented character made of two code points (character plus modifier character), the stores will not work as they will be of inconsisistant lengths.
Then you need a single line of code to tell the keyboard how you want to swap the base character for the alternate character, like this:

```
any(baseChar) + [K_BKQUOTE] &gt; index(altChar,1)
```

These 3 lines are all that you need to produce any of the 36 special characters for this language!

For those who don't understand what these lines are doing, the rule basically says, "When you type any of the base characters in the first store, followed by the BackQuote key, output the character in the same sequential position in the second store."

There are various combinations of the rule, but the stores work the same way. You can do this with a deadkey, or an activating keystroke (like semicolon in the earlier example), that gets removed from the

output.

Remember, computers don't have a mind of their own. The index() function just counts what number the character is in the first list and outputs the character in the same position in the other list (see chart below for clarity). So, if you change the order of one of the lists, you need to make sure that the characters continue to correspond the way you want them to.

1	2	3	4	5	6	7	8	9	0	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
a	A	á	Á	c	C	d	D	e	E	g	G	h	H	i	I	n	N	o	O	ó	Ó	p	P	q	Q	ø	Ø	s	S	t	T	u	U	y	Y
á	Á	ä	Ä	ç	Ç	đ	Đ	é	É	ġ	Ġ	ħ	Ħ	í	Í	ñ	Ñ	ó	Ó	ǒ	Ǒ	ṗ	Ṗ	ø	Ø	ǿ	Ǿ	ŝ	Ŝ	ŧ	Ŧ	ú	Ú	ẏ	Ẏ

Go Deeper

The Keyboard Ideas and Samples document contains sample code for doing different kinds of rules in Keyman. You can study these examples or study the Keyman language guide.