How to Modify a Keyman Keyboard

This lesson is for dealing with existing keyboards that have not yet been uploaded to keyman.com. If the project already exists on keyman.com, to edit it, you will have to pull it down from GitHub to your local computer before making modifications.

Create a folder for Keyman projects on your hard drive

In the past, Keyman files were stored in \Documents\Keyman\. Check to see if you have that directory and look at what is in there. If you have existing projects, you have two choices:

- Move them elsewhere and use the empty Keyman folder for Keyman 11+ projects
- Create a different data directory for Keyman that will only contain data managed by Keyman 11+

Starting with Keyman Developer 11, the project directory structure is in the format needed for uploading keyboards to the Keyman.com repository. It also includes the source files inside the folder so that they can be modified and updated.

Creating a project to contain it

When opening Keyman Developer, it will open the last project you were working on

1. Select **Create a New Project** from the **Project** menu
2. Choose **Basic**.
   (The option to import a Windows keyboard is about importing a Windows system keyboard to use as a starting point, not a Keyman keyboard)
3. Name the keyboard, assign copyright, version, author
4. Targets
   The default is "any". Since this includes all the others, we can leave this. Since Keyman 10, a single keyboard package can contain keyboards that will work on all major platforms.
5. Set the Path
   At the time of this tutorial, the default path was set to "Documents", which would make every keyboard a subdirectory of Documents. Not a good option! Make sure to select the directory you designated at the beginning of this document.
6. Keyboard ID
   The proposed name must be all lowercase. This is important for the Keyman.com directory structure.
7. Languages
    1. Choose Add language...
    2. Select the appropriate BCP47 (or ISO 639-3) code for each language the keyboard will support. You can usually find these codes on ethnologue.com. For Windows keyboards, the language code should also include the script. Keyman already inserts what it thinks the script is for the language you choose, but you can correct this if needed.
8. Select OK

Keyboard tab

1. Click on the Keyboards tab on the bottom of the project window
2. Open the keyboard by clicking on the blue link with the keyboard name.
3. Click on the Layout tab on the left.
   You are looking at the visual keyboard. Normally we would start here and then switch to the code view, but since we are importing an existing keyboard, we will start with the code view.
4. Open your existing keyboard file in a text editor, copy the contents and paste them at the bottom of this window.
5. Fix the header by manually merging the header information between the two files.

You can move the old comment fields if any to the top, and remove these lines (near the top) *if your original file already has them:*

```
begin Unicode > use(main) group(main) using keys
```

Save and compile

Jump ahead to the build tab on the bottom left and try to compile the keyboard. If it compiles, great! You are ready to start doing modifications. If it doesn't compile, check the warnings and fix the problems. You may get warnings about header statements being deprecated, or old code in the kmn file that is deprecated:

Messages

ibani.kmn: Compiling 'C:\Users\higby\Documents\Keyman\ibani\source\ibani.kmn' for Windows, macOS, Linux, Desktop devices... ibani.kmn: Warning: line 22 warning 209D: Header statements are deprecated; use instead the equivalent system store ibani.kmn: Error: 'C:\Users\higby\Documents\Keyman\ibani\source\ibani.kmn' was not compiled successfully for Windows, macOS, Linux, Desktop devices.

Old style headers in Keyman were written as

Legacy code

```
HOTKEY "%+I"
```

and the new style of headers is

Current code

```
store(&hotkey) "[ALT K_I]"
```

Understanding the Keyman Code

Now we are at the point where you will have to study and know something about the Keyman code.

Simple key swapping statements

It could be that your keyboard uses simple swap statements like this:

Simple key swap

```
';\' + 'i' > 'i' U+0300 c i grave
```

This statement says:

If an **i** is typed following **;** replace everything to the left of the **>** with the characters on the right. This works, but results in long and inefficient code because the comand needs to be repeated over and over for every character that you want to modify.

Using Variables and indexes

In keyboards that take advantage of the strength of the Keyman language, you will find **store**, **any** and **index** statements. The store statement creates a named variable. With variables, you can issue a single swap command that works on every character stored inside it. Here is a simple example of how these three statements are used:

Example with variables

```
store(vowelKey) 'aeiou' store(vowelAcute) 'áéíóú' any(vowelKey) + "/" > index(vowelAcute, 1)
```

In the above example, **store(vowelKey)** assigns the set of characters "**aeiou**" to the named variable "**vowelKey**". Now, you can reference the characters stored in it at any time with the **any(vowelKey)** statement.

The first two lines above assign sets of characters to two variables. The order of the characters in the set is very important! When Keyman is called to replace a character in the first variable with a character in the second variable, it counts the position of the character in the store found with the any() statement and outputs the character in the same position in the index() statement.

Index position

The number in the index statement is important because it tells it which character in the context (part that preceeds the > to get the index value from. This doesn't seem necessary in the statement above, but in the example below, the index is based on the second character in the context, so it must be set to 2:

Index in second position

```
deadkey(Acute) + any(vowelKey) > index(vowelAcute, 2)
```

For further reference

Stores are best explained on the Keyman website here:

Stores, any(), and index()[1]

The Keyman website has a complete code reference that you will have to get familiar with. Read up on how the code works.

Keyman Developer Keyboard Tutorial[2]

Adding a character to an existing keyboard

The most common reason we get asked to edit an existing keyboard is to add a character that is missing. We will include an example here of the Ibani keyboard which wants the addition of a dot below the letter **d**. In Unicode, the desired character is U+0323 Combining dot below.

This is quite an easy assignment because the keyboard already contains a keystroke sequence to produce the dot below on **b** and **s**.

Ibani keyboard coding exercise

We won't show the whole original keyboard -- as it is not the most efficient coding. Here is enough for you to observe and discover where and how to insert the additional code to make the dot appear under both upper and lowercase **d**.

Ibani keyboard before adding dot below d

```
begin Unicode > use(Main) store(Let2BMod) "cdghkmnyzCDGKLMNXY$?*" store(ModLet) "çɗgɦƙɲŋƴʒÇƊɣƘ£ƝŊƎ₦?°" store(Let2BDot) "bBsS" store(vowel) "aeiouAEIOU" store(vAcute) "áéíóúÁÉÍÓÚ" store(vMid) "āēīōūĀĒĪŌŪ" store(vGrave) "àèìòùÀÈÌÒÙ" store(vCircum) "âêîôûÂÊÎÔÛ" store(vCaron) "ǎěǐǒǔǍĚǏǑǓ" store(vDot) "ạẹịọụẠẸỊỌỤ" store(nasal) "mnMN" store(nAcute) "ḿńḾŃ" store(LetN) "nN" store(nGrave) "ǹǸ" store(nCaron) "ňŇ" group(main) using keys c deadkeys are identified c single deadkeys + ";" > deadkey(modlet) c applies to dotted vowels and Letters to be modified + "'" > deadkey(acute) + "-" > deadkey(mid) + "`" > deadkey(grave) + "^" > deadkey(circum) + "&" > deadkey(caron) + "!" > deadkey(downstep) c single deadkeys are cleared when typed twice (dk acute has a second use here) dk(modlet) + ";" > ";" dk(acute) + "'" > U+02C8 c modifier letter vertical line (') U+02C8 + "'" > "'" c just an acute accent dk(mid) + "-" > "-" dk(grave) + "`" > "`" dk(circum) + "^" > "^" dk(caron) + "&" > "&" dk(downstep) + "!" > "!" c 1st set (;) combo deadkeys for tones on dotted vowels (order does not matter) dk(modlet) + "'" > deadkey(dotAcute) dk(acute) + ";" > deadkey(dotAcute) dk(modlet) + "-" > deadkey(dotMid) dk(mid) + ";" > deadkey(dotMid) dk(modlet) + "`" > deadkey(dotGrave) dk(grave) + ";" > deadkey(dotGrave) dk(modlet) + "^" > deadkey(dotCircum) dk(circum) + ";" > deadkey(dotCircum) dk(modlet) + "&" > deadkey(dotCaron) dk(caron) + ";" > deadkey(dotCaron) dk(modlet) + "!" > deadkey(dotDownstep) dk(downstep) + ";" > deadkey(dotDownstep) c assignments c plain letters are modified, vowels are dotted dk(modlet) + any(Let2BMod) > index(ModLet,2) dk(modlet) + any(vowel) > index(vDot,2) dk(modlet) + any(Let2BDot) > index(Let2BDot,2) U+0323 c plain vowels & nasals are tone marked dk(acute) + any(vowel) > index(vAcute,2) dk(acute) + any(nasal) > index(nAcute,2) dk(mid) + any(vowel) > index(vMid,2) dk(mid) + any(nasal) > index(nasal,2) U+0304 dk(grave) + any(vowel) > index(vGrave,2) dk(grave) + any(LetN) > index(nGrave,2) dk(circum) + any(vowel) > index(vCircum,2) dk(circum) + any(nasal) > index(nasal,2) U+0302 dk(caron) + any(vowel) > index(vCaron,2) dk(caron) + any(LetN) > index(nCaron,2) dk(downstep) + any(vowel) > index(vowel,2) U+030B dk(downstep) + any(nasal) > index(nasal,2) U+030B
```

Observations about the above code

The key sequence **;** + **s** produces **ș** -- so it would be easy to add **d** to the same store.
However, **;** + **d** is *already used* to produce this character: **ɗ**
What can we do? Fortunately the hooked d is not needed in the Ibani orthography and may be removed from the code.

The semicolon is pressed and becomes **deadkey(modlet)** in **line 22**.

deadkey(modlet) is used to produce the dot below the s and b in **line 58.**

deadkey(modlet) is also used on **store(ModLet)** to swap with **store(Let2BMod)** in **line 56**

You can try to solve this by pasting the above code into Keyman Developer (or if following in class, use the file provided).

When you think you have found the solution, click below to compare your results

Click to see the solution[3]

d, **D** and their swap characters **ɗ**, **Ɗ** must be removed from the Let2BMod and ModLet stores as follows: store(Let2BMod) "cghkmnyzCGKLMNXY$?*" store(ModLet) "çgɦƙɲŋyʒÇɣK£ŊŊƎY₦ʔ°"

Then **d** and **D** must be added to the Let2BDot store: store(Let2BDot) "bBsSdD"

Once you figure out what the code is doing, it is easy!

Your job is finished in so far as your client will be happy! However, we have only done the minimum that was required. Do you want one happy client or do you want a thousand happy clients? For that to happen, we'll have to prepare this file and its documentation for uploading to Keyman.com

---

[1] https://help.keyman.com/developer/11.0/guides/develop/tutorial/step-6

[2] https://help.keyman.com/developer/11.0/guides/develop/tutorial/

[3] https://lingtran.net/#wpfade-1-inner