Customizing the Keypad Keymap

by John Orr and Carolyn Scheppner

The article ‘‘Loading Keymaps’’ in this issue of Amiga Mail discusses how to load a keymap file from disk and add it to the system’s list of keymaps. The article also mentions that, although it does have some legitimate uses, a console-based application rarely needs to concern itself with system-global keymaps. In most cases, it is both easier and more system friendly for the application to duplicate its console’s keymap and make some modifications to the copy of the keymap. Since such an application has a private copy of the keymap, the application does not have to interfere with the user’s global keymap settings. Since this application is using a copy of the user’s default keymap, the technique works with many different national keyboards.

The KeyMap

To understand how to alter a KeyMap, you have to understand the layout of a KeyMap. This is covered in the ‘‘Keymap Library’’ chapter of the *Amiga ROM Kernel Reference Manual: Libraries*, although it is a bit hard to follow. Hopefully, this section describes the layout of an Amiga Keymap a little better.

Each key on the Amiga keyboard has its own 7-bit raw key code. It is a raw key because there is no character value assigned to it yet, so it isn’t ‘‘cooked’’ yet. The Amiga doesn’t bother assigning a character value to a specific key on a keyboard because the letter printed on a key can vary greatly between keyboards intended for use with different languages. The Amiga uses a *KeyMap* to map raw key codes from the keyboard to an ANSI character.

The Amiga’s input.device handles generating raw key input events. The input.device reads the state of the keyboard using the keyboard.device. There are two ways to use the keyboard to trigger the input.device into generating a raw key input event. When the user presses a key, the input.device creates an `IECLASS_RAWKEY` InputEvent structure (see <*devices/inputevent.h*>) and stores the raw key code as a byte in the ie_Code field. This is known as a ‘‘key down’’ event. The input.device also creates a similar input event when the user releases a key, but it sets the high bit of the raw code byte to indicate the user let go of the key (sometimes called a ‘‘key up’’ event). If any qualifier keys (Control, Shift, or Alt) were pressed when the raw key was pressed (or released), the input.device sets some qualifier bits in the InputEvent’s ie_Qualifier field. The input.device then adds the InputEvent to the input stream.

As the input events travel down the input stream, a variety of input handlers (like Intuition’s input handler) have the opportunity to examine the input events, possibly adding and removing input events. If the `IECLASS_RAWKEY` InputEvent eventually arrives at the console.device’s input handler, the console.device can use a KeyMap to convert (or ‘‘cook’’) the raw key event into one of several things:

An ANSI character
A String (32 characters or less)
A Dead Key (or Double Dead Key)

The following is the KeyMap structure (from *<devices/keymap.h>*):

```
struct   KeyMap
{
    UBYTE   *km_LoKeyMapTypes;
    ULONG   *km_LoKeyMap;
    UBYTE   *km_LoCapsable;
    UBYTE   *km_LoRepeatable;
    UBYTE   *km_HiKeyMapTypes;
    ULONG   *km_HiKeyMap;
    UBYTE   *km_HiCapsable;
    UBYTE   *km_HiRepeatable;
};
```

The KeyMap maps the range from raw key 0x00 through 0x7F. As mentioned earlier, the high bit of the raw code is reserved for the up/down state of the key. Each field in the KeyMap structure is a vector that points to a table. The KeyMap splits the range of raw keys into a low set, 0x00 through 0x39, and a high set, 0x40 though 0x7F. The vectors with names that start with ‘‘km_Lo’’ refer to the lower range (0x00 through 0x39). Likewise, the vectors with names that start with ‘‘km_Hi’’ refer to the higher range (0x40 through 0x7F). Each table has an entry for each raw key in its range.

The km_LoKeyMapTypes and km_HiKeyMapTypes fields each point to an array of bytes. Each array has an entry for each raw key in its range (0x00-0x39 for LoKeyMapTypes and 0x40-0x7F for HiKeyMaps). These arrays, known as the *Key Type tables*, describe a key’s type and its relevant qualifiers. Each entry is a bitfield composed of the following flags (from *<devices/keymap.h>*):

```
KCF_SHIFT    /* The mapping is different for shifted mapping vs. non-shifted */
KCF_ALT      /* The mapping is different for ALTed keys vs. non-ALTed */
KCF_CONTROL  /* The mapping is different for CTRLed keys vs. non-CTRLed */

KCF_DEAD    /* This key is a ‘‘deadable’’ key */
KCF_STRING  /* This key maps to a string */
KCF_NOP     /* This key maps to nothing */
```

The km_LoKeyMap and km_HiKeyMap fields each point to an array containing a long word entry for each raw key in its range. These arrays are known as the *Key Map* tables. The purpose of the long word depends on the corresponding entry from the key type tables:

For string keys (the KCF_STRING bit from the key type table is set), the entry is a 32-bit pointer to string data (more on this later).

For deadable keys (the KCF_DEAD bit from the key type table is set), the entry is a 32-bit pointer to dead key data.

For KCF_NOP keys, the entry is undefined.

For all other keys (for lack of a better term, we'll call them ‘‘normal’’ keys), the entry consists of four bytes. Each of these bytes is an eight bit ANSI code. The raw key ‘‘maps’’ to one of these four bytes. The qualifier keys that were pressed when the user hit the raw key determine which of the four ANSI codes to use.

Unfortunately, since each normal raw key can only map to one of four bytes, there is not enough space to handle all possible qualifier combinations. Since there are three qualifier keys (Shift, Alt, and Control), each raw key would require eight bytes to have an entry for all possible qualifier key combinations.

As this scheme can't support all possible qualifier combinations, it only supports some combinations. The easiest case to handle is where the user hits the raw key without any qualifiers. The raw key without qualifiers always maps to the ANSI value in the low order byte of its entry in the Key Map table.

The meaning of the upper three bytes of the key map table entry changes according to the raw key's key type value (from the key type table). The following chart shows how the key type value changes the meaning of all four bytes. If a qualifier does not appear in the ‘‘Key Type’’ column, the keymapping software ignores that qualfier for the key.

Table 1 - ‘‘Normal’’ Raw Keys Lo/HiKeyMap Table Entries

Key Type	Four-Byte Entry from Key Map Table:			
	bits 31-24	bits 23-16	bits 15-8	bits 7-0
no qualifiers	-	-	-	raw key alone
KCF_SHIFT	-	-	Shift	raw key alone
KCF_ALT	-	-	Alt	raw key alone
KCF_CONTROL	-	-	Ctrl	raw key alone
KCF_SHIFT|KCF_ALT	Shift+Alt	Alt	Shift	raw key alone
KCF_CONTROL|KCF_ALT	Ctrl+Alt	Ctrl	Alt	raw key alone
KCF_CONTROL|KCF_SHIFT	Ctrl+Shift	Ctrl	Shift	raw key alone
KC_VANILLA	Shift+Alt	Alt	Shift	raw key alone*

*If the Control key is down, the raw key maps to the low-byte entry (even if other qualifier keys are down), but the fifth and sixth bits of the ANSI value are cleared.

A dash ('-') indicates the value in that byte is not used.

For example, if the key type is KCF_SHIFT|KCF_ALT and the user is holding the shift and alt keys while pressing a normal raw key, the raw key maps to the ANSI code in the high order byte (bits 31-24). If the key map entry is the long word 0xD0F04464, the raw key maps to the ANSI value 0xD0.

Another good example to consider is where the key type is KC_VANILLA. *The vanilla qualifier does not mean plain!* KC_VANILLA is a combination (a logical OR) of the three qualifier key types. When the key type is KC_VANILLA and the user is holding the Control key, the keymapping software maps the the raw key to the low order byte, but it clears bits five and six of the value it ouputs. The keymapping software ignores the state of the shift and Alt keys. Using the key map entry 0xD0F04464, the raw key maps to the low byte (0x64) but with its fifth and sixth bit cleared:

```
%0110 0100    This is 0x64 in binary
%1001 1111   bitwise AND to clear bits 5 and 6
-----------
%0000 0100    = 0x04
```

The next fields from the KeyMap structure are km_LoCapsable and km_HiCapsable, which point to the Capsable tables. These tables contain a single bit for each raw key in their range. Their purpose is to tell the keymapping software what to do when the Caps Lock button is on (the LED in the Caps Lock button is lit). If a raw key's Capsable bit is set, the keymapping software treats the raw key as a shifted raw key while the Caps Lock LED is on.

Like the Capsable tables, the Repeatable tables (km_LoRepeatable and km_HiRepeatable) contain a single bit for each raw key in their range. If a raw key's Capsable bit is set, the keymapping software should allow the key to repeat if the user holds it down long enough. The Input Preferences editor sets the time intervals that the input.device waits before repeating the key and between key repeats. The input.device takes care of adding these events to the input stream. When the keymapping software sees the series of repeated keys, it may elect to throw away almost all of them if the key is not marked as repeatable in the keymap.

String Output Keys

A keymap can also map a raw key to a string. A raw key maps to a string if the key's entry in the key type table has the KCF_STRING bit set. For string keys, the corresponding long word entry in the Lo/HiKeyMap is a 32-bit pointer to the string data.

The string data consists of a table of 16-bit entries. Unlike the ‘‘normal’’ keys, string keys can account for every possible qualifier key combination. Since there must be an entry in the string key's table for each qualfier combination, the number of table entries depends on how many qualifier bits are set in the key's key type table entry. If x is the number of qualifier bits set in the key type table then the number of table entries is 2^x.

If all three qualifier bits are set, there are eight entries. They appear in the table in the following order:

Table 2 - String Key Data Table

```
000  No Qualifiers
001  Shift
010  Alt
011  Alt+Shift
100  Control
101  Control+Shift
110  Control+Alt
111  Control+Alt+Shift
```

To adjust the table above for fewer qualifiers, remove the rows that contain the missing qualifier. For example, if the KCF_SHIFT and KCF_CONTROL bits are set, the KCF_ALT qualifier is missing, so remove all the lines with an ‘‘Alt’’ in them:

Table 3 - A Shiftable Control Key

```
00  No Qualifiers
01  Shift
10  Control
11  Control+Shift
```

Each entry in the string data table is organized into two bytes, the first byte tells the keymapping software the length of the string that the raw key maps to. The second byte is a positive offset from the beginning of the table to the string itself. Since this offset is only eight bits wide, every string must start within the 255 bytes after the beginning of the table. For a raw key that uses a full string data table, the keymap needs the first 16 bytes, leaving enough room for about 34 bytes per string. Normally, the key’s strings should not exceed 32 bytes as larger strings can overflow the console.device’s input buffer, so 32 bytes is a good upper limit.

Unlike a C-style string, the raw key’s string does not need a zero terminator at the end (although it’s not a bad idea to have a zero there if there is enough space).

Simple KeyMap Example

The following is a pseudo-assembler fragment that shows the key type and the map table entries for the first three keys of the high keymap (raw keys 0x40, 0x41, and 0x42):

```
newHiKeyTypes:
    DC.B        KCF_ALT,KC_NOQUAL,      ;key 0x40 (spacebar), key 0x41 (backspace)
    DC.B        KCF_STRING+KCF_SHIFT,   ;key 0x42 is the tab key on the US keyboard
        ...     ;(other keys)
newHiKeyMap:
    DC.B        0,0,$A0,$20     ;key 0x40: nil, nil, the Alted-space key maps to $A0 which
                                ;  is the ANSI Non-Breakable Space (NBSP), the plain space
                                ;  key maps to $20 which is the space character.
    DC.B        0,0,0,$08       ;key 0x41: Back Space key only
    DC.L        newkey42        ;key 0x42: definition for string to output for Tab key
        ...                     ;(other keys)
newkey42:
    DC.B        new42ue - new42us       ;length of the unshifted string (new42us)
    DC.B        new42us - newkey42      ;number of bytes from start of
```

```
                                      ;  string descriptor to start of this string
    DC.B        new42se - new42ss     ;length of the shifted string (new42ss)
    DC.B        new42ss - newkey42    ;number of bytes from start of
                                      ;  string descriptor to start of this string
new42us:  DC.B        '[TAB]'
new42ue:
new42ss:  DC.B        '[SHIFTED-TAB]'
new42se:
```

The keymap fragment above shows two kinds of keys, a ‘‘normal’’ key and a string key. Looking at the key types table which starts at the newHiKeyTypes label, key 0x40 is neither a dead key, a No Op key, or a string key, so key 0x40 is a ‘‘normal’’ key. It has one modifier, `KCF_ALT`. Looking at Table 1, the Key Map table entry for key 0x40 should only have two defined bytes. The third byte is the raw key with the Alt modifier and the fourth byte is the raw key without modifiers. Looking at key 0x40’s entry in the key map table (the first four bytes of newHiKeyMap), if the user hits raw key 0x40 with the Alt key down, this keymap maps the key to ANSI character 0xA0, which is the ANSI Non-Breakable Space character (NBSP). Since only the `KCF_ALT` qualifier bit is set, this keymap ignores the other qualifier keys.

Key 0x41 is similar to key 0x40 as it is a ‘‘normal’’ key as well. One way in which key 0x41 differs from key 0x40 is that it has no qualifiers. According to Table 1, if the user presses raw key 0x41 with or without qualifiers, this keymap maps to the fourth byte in key 0x41’s entry in the key map table, so raw key 0x41 maps to the ANSI character 0x08, which is the backspace character.

The last key, 0x42, has two bits set in its entry from the key types table, `KCF_STRING` and `KCF_SHIFT`, so it is a shiftable string key. For a string key, the entry from the key map table is a 32-bit pointer to string key data instead of the four ANSI codes a ‘‘normal’ key uses. Key 0x42’s key map entry points to newkey42. As key 0x42 is shiftable, the string data should only have two entries in it. To adjust Table 2 for the shift qualifier, remove the rows that contain Alt and Control. The table that starts at newkey42 should look like this:

```
00   No Qualifiers
01   Shift
```

Looking at the keymap example, the ‘‘No Qualifiers’’ entry in the string table shows that the length of the ‘‘No Qualifiers’’ string is new42ue-new42us. The second byte in the string table entry is the offset from newkey42 to the ‘‘No Qualifiers’’ string, which points to the string ‘‘[TAB]’’.

A Word About Dead-Class Keys

The Amiga keymap scheme also supports dead-class keys, which allow one key press to modify the next key press. However, robustly altering a keymap by adding dead-class keys requires an impractical amount of overhead and is not particularly useful for the purposes described here. Dead-class keys are not covered in this article. For more information, see the ‘‘Keymap Library’’ chapter of the *Amiga ROM Kernel Reference Manual: Libraries*.

Cloning and Altering the KeyMap

The AppKeyMap code at the end of this article contains a function which duplicates a KeyMap structure plus all the KeyMap's associated tables. Cloning a keymap is a fairly straightforward memory copy, although the AppKeyMap code adds a little twist to make it easier for other functions to reference the entries in the table. The way <*devices/keymap.h*> defines the KeyMap structure, the keymap tables are split into a low set (‘‘km_Lo...’’) and a high set (‘‘km_Hi...’’). This makes looking up entries in the tables a nuisance because you have to treat ‘‘Lo’’ raw key and ‘‘Hi’’ raw keys separately. When the CreateAppKeyMap() function creates the tables, it organizes the tables so the ‘‘Hi’’ table directly follows the ‘‘Lo’’, joining the two tables into one. This allows other functions in the AppKeyMap code to use a raw key value as an index into the table.

Altering the keymap is also rather straightforward, particularly because the CreateAppKeyMap() function made the table entries much easier to reference. The only remotely complicated part is handling the Capsable and Repeatable tables. These are both bit tables (there a single bit entry for each raw key), so handling them in C is a little cumbersome.

About the Code Example

The *appmap_demo.c* example is a shell-only program. It opens the console.device and uses its `CD_ASKKEYMAP` command to obtain a copy of the shell's current keymap. Using the AppKeyMap functions, the example duplicates and alters the keymap copy. *Appmap_demo.c* then uses the console's `CD_SETKEYMAP` command to install the keymap copy as the shell's current keymap. The user can type at the shell's window trying out the altered keymap. When finished, the example restores the shell's original keymap using the console's `CD_SETKEYMAP` command and exits.

An application can also use this technique when attaching a console.device to an Intuition window.