Adding New Capabilities to Kiibohd

Written by J David Smith
Published on 13 January 2016

One of the reasons I bought my ErgoDox was because I'd be able to hack on it. Initially, I stuck to changing the layout to Colemak and adding bindings for media keys. Doing this with the Kiibohd firmware is reasonably straightforward: clone the repository, change or add some `.kll` files[1], and then recompile and reflash using the provided shell scripts.

This week, I decided to finally add a new capability to my board: LCD status control. One thing that has irked me about the Infinity ErgoDox is that the LCD backlights remain on even when the computer is off. As my computer is in my bedroom, this means that I have two bright nightlights unless I unplug the keyboard before going to bed.

Fortunately, the Kiibohd firmware and KLL language support adding *capabilities*, which are C functions conforming to a couple of simple rules, and exposing those capabilities for keybinding. This is how the stock Infinity ErgoDox LCD and LED control is implemented, and how I planned to implement my extension. However, the process is rather poorly documented and presented some unexpected hurdles. Ultimately, I got it working and wanted to document the process for posterity here.[2] The rest of this post will cover in detail how to add a new capability `LCDStatus(status)` that controls the LCD status.[3]

Background & Setting Up

Before attempting to add a capability, make sure you can compile the stock firmware and flash it to a keyboard successfully. The instructions on the

Kiibohd repository are solid, so I won't reproduce them here.

An important note before beginning is that it is possible to connect to the keyboard via a serial port. On Linux, this is typically `/dev/ttyACM0`. The command `screen /dev/ttyACM0` as root will allow one to connect, issue commands, and view debug messages during development.

This post is *specifically* concerned with implementing an `LCDStatus` capability. If you don't have an LCD to control the status of, then this obviously will be nonsense. However, much of the material (e.g. on states and state types) may still be of use.

The Skeleton of a Capability

Capabilities in Kiibohd are simply C functions that conform to an API: `void` functions with three parameters: `state`, `stateType`, and `args`. At the absolute minimum, a capability will look like this:

```
void my_capability(uint8_t state, uint8_t stateType, uint8_t *args) {
}
```

The combinations of `state` and `stateType` describe the keyboard state:[4]

`stateType`	`state`	meaning
0x00 (Normal)	0x00	key depressed
0x00 (Normal)	0x01	key pressed
0x00 (Normal)	0x02	key held
0x00 (Normal)	0x03	key released
0x01 (LED)	0x00	off
0x01 (LED)	0x01	on
0x02 (Analog)	0x00	key depressed
0x02 (Analog)	0x01	key released
0x02 (Analog)	0x10 - 0xFF	Light Press - Max Press
0x03-0xFE		Reserved
0xFF (Debug)	0xFF	Print capability signature

Every capability should implement support for the debug state. Without this, the capability will not show up in the `capList` debug command.

```
void my_capability(uint8_t state, uint8_t stateType, uint8_t *args) {
    if ( state == 0xFF && stateType == 0xFF ) {
        print("my_capability(arg1, arg2)");
        return;
    }
}
```

Within this skeleton, you can do whatever you want!
The full power of C is at your disposal, commander.

Turning Out the Lights

The LCD backlights have three channels corresponding to the usual red, green, and blue. These are unintuitively named `FTM0_C0V`, `FTM0_C1V`, and `FTM0_C2V`.[5] To turn them off, we simply zero them out:

```
void LCD_status_capability(uint8_t state, uint8_t stateType, uint8_t *args) {
    if ( state == 0xFF && stateType == 0xFF ) {
        print("my_capability(arg1, arg2)");
        return;
    }
    FTM0_C0V = 0;
    FTM0_C1V = 0;
    FTM0_C2V = 0;
}
```

With this addition, we have a capability that adds new functionality! I began by adding this function to `Scan/STLcd/lcd_scan.c`, because I didn't and still don't want to mess with adding new sources to CMake. Now we can expose this simple capability in KLL:

```
LCDStatus => LCD_status_capability();
```

It can be bound to a key just like the built-in capabilities:

```
U"Delete": LCDStatus();
```

If you were to compile and flash this firmware, then pressing *Delete* would now turn off the LCD instead of deleting. [6]

Adding Some Arguments

The next step in our quest is to add the `status` argument to the capability. This is pretty straightforward. First, we will update the KLL to reflect the argument we want:

```
LCDStatus => LCD_status_capability( status : 1 );
```

The `status : 1` in the signature defines the *name* and *size* of the argument in bytes. The name isn't used for anything, but should be named something reasonable for all the usual reasons.

Then, our binding becomes:

```
U"Delete": LCDStatus( 0 );
```

Processing the arguments in C is, unfortunately, a bit annoying. The third parameter to our function (`*args`) is an array of `uint8_t`. Since we only have one argument, we can just dereference it to get the value. However, there are examples of more complicated arguments in `lcd_scan.c` illustrating how not-nice it can be.

```
void LCD_status_capability(uint8_t state, uint8_t stateType, uint8_t *args) {
    if ( state == 0xFF && stateType == 0xFF ) {
        print("my_capability(arg1, arg2)");
    }

    uint8_t status = *args;
    if ( status == 0 ) {
        FTM0_C0V = 0;
        FTM0_C1V = 0;
        FTM0_C2V = 0;
```

```
        }
}
```

Figuring out how to restore the LCD to a reasonable state is less straightforward. What I chose to do for my implementation was to grab the last state stored by `LCD_layerStackExact_capability` and use that capability to restore it.[7] `layerStackExact` uses two variables to track its state:

```
uint16_t LCD_layerStackExact[4];
uint8_t LCD_layerStackExact_size = 0;
```

It also defines a struct which it uses to typecast the `*args` parameter.

```
typedef struct LCD_layerStackExact_args {
        uint8_t numArgs;
        uint16_t layers[4];
} LCD_layerStackExact_args;
```

We can turn the LCD back on by calling the capability with the stored state. Note that I *copied* the array, just to be safe. I'm not sure if it is necessary but I didn't want to have to try to debug corrupted memory.

```
void LCD_status_capability(uint8_t state, uint8_t stateType, uint8_t *args) {
    if ( state == 0xFF && stateType == 0xFF ) {
        print("my_capability(arg1, arg2)");
    }

    uint8_t status = *args;
    if ( status == 0 ) {
        FTM0_C0V = 0;
        FTM0_C1V = 0;
        FTM0_C2V = 0;
    } else if ( status == 1 ) {
        LCD_layerStackExact_args stack_args;
        stack_args.numArgs = LCD_layerStackExact_size;
        memcpy(stack_args.layers, LCD_layerStackExact, sizeof(LCD_layerStackExact
```

```
        LCD_layerStackExact_capability( state, stateType, (uint8_t*)&stack_args )
    }
}
```

Now binding a key to `LCDStatus(1)` would turn *on* the LCDs.

Creating Some State

Like most mostly-functional programmers, I abhor state. Don't like it. Don't want it. Don't want to deal with it. However, if we want to implement a toggle that's exactly what we'll need. We simply create a global variable (*ewww*, I know! But we can deal) `LCD_status` and set it to the appropriate values. Then toggling is as simple as making a recursive call with `!LCD_status`.

```
uint8_t LCD_status = 1; // default on
void LCD_status_capability(uint8_t state, uint8_t stateType, uint8_t *args) {
    if ( state == 0xFF && stateType == 0xFF ) {
        print("my_capability(arg1, arg2)");
    }

    uint8_t status = *args;
    if ( status == 0 ) {
        FTM0_C0V = 0;
        FTM0_C1V = 0;
        FTM0_C2V = 0;
        LCD_status = 0;
    } else if ( status == 1 ) {
        LCD_layerStackExact_args stack_args;
        stack_args.numArgs = LCD_layerStackExact_size;
        memcpy(stack_args.layers, LCD_layerStackExact, sizeof(LCD_layerStackExact
        LCD_layerStackExact_capability( state, stateType, (uint8_t*)&stack_args )
        LCD_status = 1;
    } else if ( status == 2 ) {
        status = !LCD_status;
        LCD_status_capability( state, stateType, &status );
    }
}
```

Binding a key to `LCDStatus(2)` will now…do nothing (probably). Why? The problem is that the

capability will *continuously* fire while the key is held down, and the microcontroller is plenty fast enough to fire arbitrarily many times during even a quick tap. So, we will guard the toggle[8] with an additional condition:

```
else if ( status == 2 && stateType == 0 && state == 0x03 ) {
    // ...
}
```

Release (`0x03`) is the only state that fires only once, so we check for that. Alas, even after fixing this, we still only have one LCD bent to our will! What about the other?

Inter-Keyboard Communication

The two halves of an Infinity ErgoDox are actually completely independent and may be used independently of one another. However, if the two halves are connected then they can communicate by sending messages back and forth.[9] I haven't delved into the network code, but I assume it is probably serial like the debug communication.

Very Important Note: You *must* flash both halves of the keyboard to have matching implementations of the capability when using communication.

The code to communicate changes relatively little from case to case but is rather long to reconstruct by hand. Therefore, I basically just copied it from `LCD_layerStackExact_capability`, changed the function it referred to, and called it a day. Wonderfully, that worked! [10]

```
#if defined(ConnectEnabled_define)
  // Only deal with the interconnect if it has been compiled in
  if ( status == 0 || status == 1 ) {
     // skip in the recursive case

    if ( Connect_master )
      {
        // generatedKeymap.h
```

```
            extern const Capability CapabilitiesList[];

            // Broadcast LCD_status remote capability (0xFF is the broadcast id)
            Connect_send_RemoteCapability(
                0xFF,
                LCD_status_capability_index,
                state,
                stateType,
                CapabilitiesList[ LCD_status_capability_index ].argCount,
                &status);
        }
    }
#endif
```

The magic constant `LCD_status_capability_index` ends up available through some build magic that I haven't delved into yet.

The Final Result

Putting all of that code together, we get:

```
uint8_t LCD_status = 1; // default on
void LCD_status_capability(uint8_t state, uint8_t stateType, uint8_t *args) {
    if ( state == 0xFF && stateType == 0xFF ) {
        print("my_capability(arg1, arg2)");
    }

    uint8_t status = *args;
    if ( status == 0 ) {
        FTM0_C0V = 0;
        FTM0_C1V = 0;
        FTM0_C2V = 0;
        LCD_status = 0;
    } else if ( status == 1 ) {
        LCD_layerStackExact_args stack_args;
        stack_args.numArgs = LCD_layerStackExact_size;
        memcpy(stack_args.layers, LCD_layerStackExact, sizeof(LCD_layerStackExact
        LCD_layerStackExact_capability( state, stateType, (uint8_t*)&stack_args )
        LCD_status = 1;
    } else if ( status == 2 && stateType == 0 && state == 0x03 ) {
        status = !LCD_status;
        LCD_status_capability( state, stateType, &status );
    }

#if defined(ConnectEnabled_define)
```

```
// Only deal with the interconnect if it has been compiled in
if ( status == 0 || status == 1 ) {
   // skip in the recursive case

  if ( Connect_master )
    {
      // generatedKeymap.h
      extern const Capability CapabilitiesList[];

      // Broadcast LCD_status remote capability (0xFF is the broadcast id)
      Connect_send_RemoteCapability(
          0xFF,
          LCD_status_capability_index,
          state,
          stateType,
          CapabilitiesList[ LCD_status_capability_index ].argCount,
          &status);
        }
    }
#endif
}
```

I have a working implementation of this on my fork of kiibohd. I'm looking forward to adding more capabilities to my keyboard now that I've gotten over the initial learning curve. I've already got a couple in mind. Generic Mod-key lock, anyone?