How to Connect a Keypad to the BeagleBone Black

Gary Williams

Marc DeRosier

Conor Brady

Jong-Ju Park

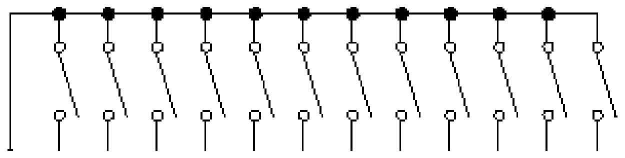
CMPT 433

Fall 2014

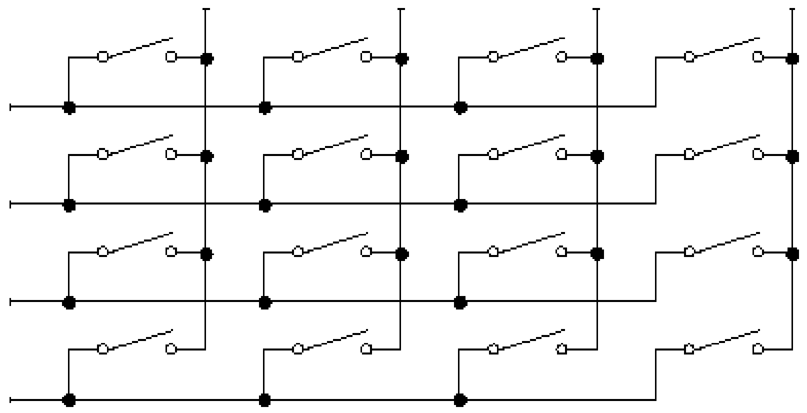
Keypads provide a simple and convenient mechanism for a user to interact with an embedded device. They are available in a variety of form factors (e.g., membrane, conductive rubber, and pushbutton) and electrical arrangements (e.g., common-bus and matrix) (see fig. 1).This guide covers the use of a common-bus style keypad.

Fig. 1: Common Keypad Formats

12-Key Shared-Bus Keypad

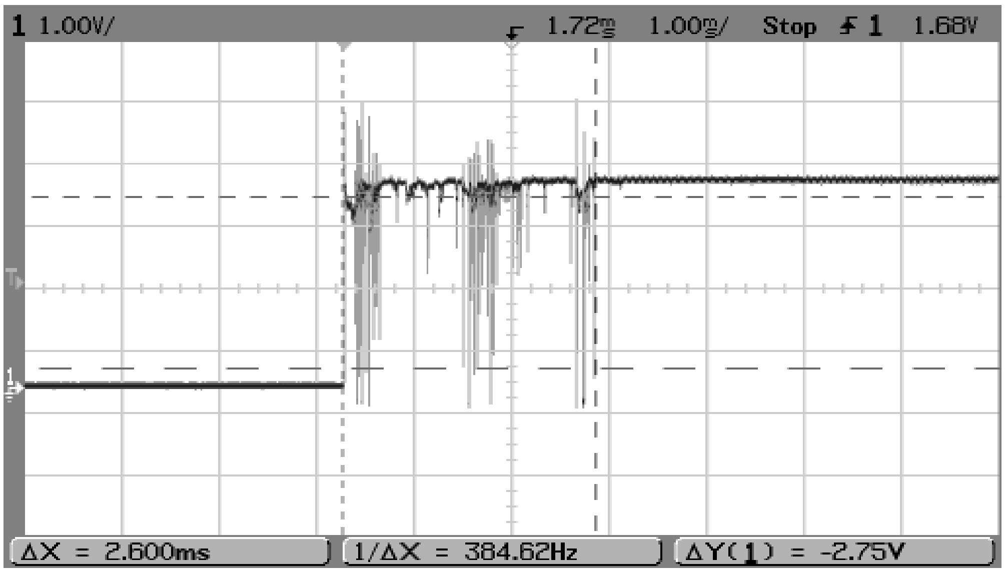


16-Key Matrix Keypad



One problem with using mechanical switches is that they will not produce perfectly-clean digital signals; instead of transitioning from logical 0 to logical 1 instantaneously, the signal will bounce erratically for a few milliseconds. This phenomenon is called "noise" (see fig. 2--image source: Wikipedia).

Fig. 2: Digital Noise

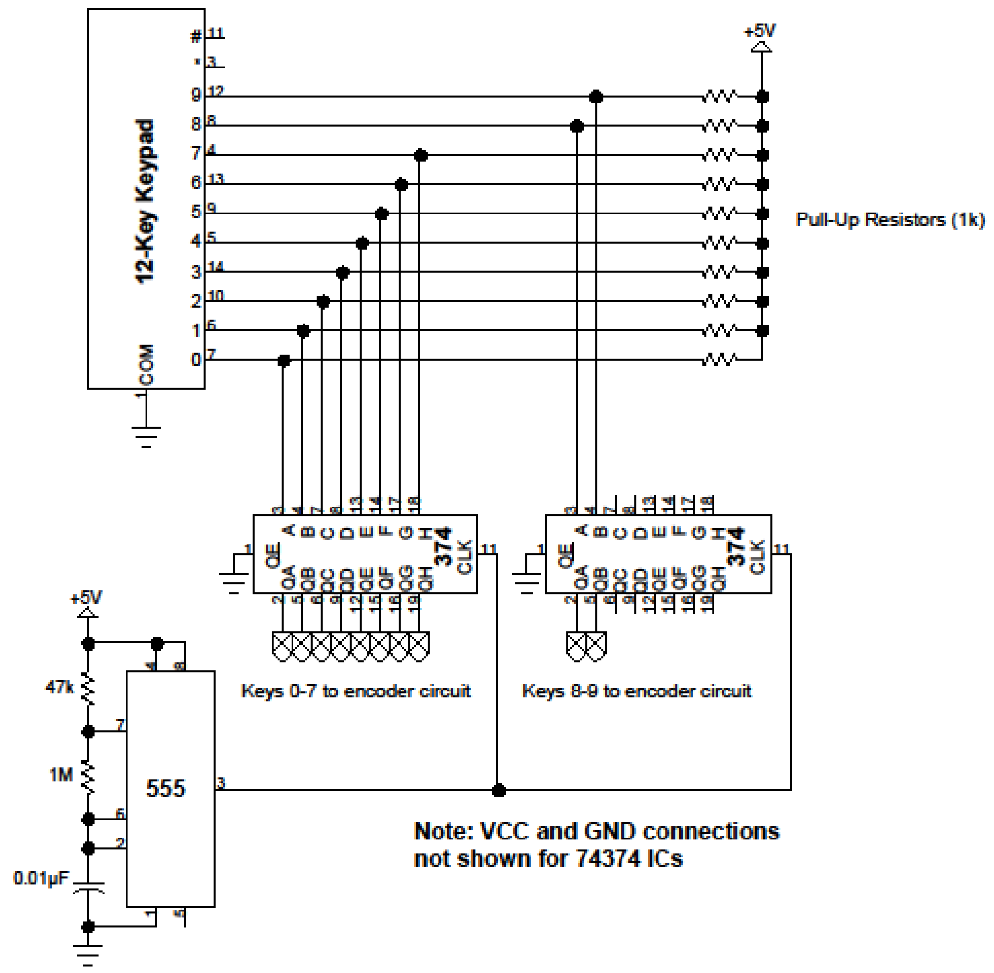


If unmitigated, digital noise will cause a number of problems for electronics; for example, a single keypress could trigger a hardware interrupt more than once. If the state of a digital signal is read during a noisy transition, the resulting measurement will be unpredictable.

We can mitigate digital noise by using a process called "debouncing". The goal of debouncing is to make the signal transition more distinct and reliable. Debouncing can be done either in software or in hardware. Software-based debouncing involves sampling a signal at timed intervals, either continuously or just after the start of a transition is detected, and waiting for the signal to stabilize before acting upon it. However, this approach introduces complexity and overhead to software, particularly for a low-powered embedded device that may be monitoring multiple signals concurrently.

This guide will cover hardware debouncing, which presents a cleaner, more reliable interface to the software. One approach to hardware-based debouncing is to use clocked latches; a clock signal triggers some D-type latches (one per signal being monitored) to record the state of the signals at regular intervals. We tried this approach first because we happened to have the components needed to implement it; see fig. 3 for our implementation.

Fig. 3: Keypad Debouncer Circuit using Clocked Latches


12-Key Keypad
COM
+5V
Pull-Up Resistors (1k)
374
374
555
47k
1M
0.01µF
Keys 0-7 to encoder circuit
Keys 8-9 to encoder circuit
Note: VCC and GND connections not shown for 74374 ICs

There are several important things to note about this circuit. First, the pin numbers on the keypad itself are not in an intuitive order, and they're specific to the model of keypad that we happened to acquire. Second, pull-up resistors are used so that the key signals float high, as our encoder circuit happens to require active-low signals. We used ten 1/4-watt resistors, but it would be simpler (and more compact) to use what's called a "resistor network", a component that consolidates an array of resistors into one package with one end of each resistor connected together.

This circuit uses the 555 timer IC (also available as NE555, LM555, etc.) in what's known as an "astable" (self-triggering) configuration. The frequency of the clock is given by the equation

$$F = \frac{1.44}{(R1 + 2R2)C1}$$

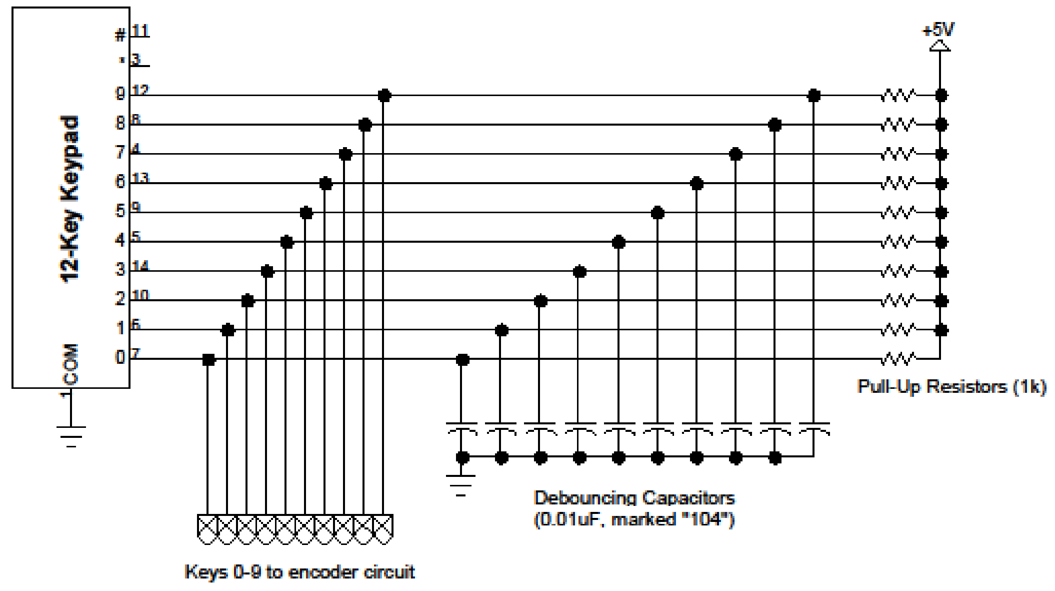
Where R1 is the value of the resistor that connects pin 7 to +5v, R2 is the value of the resistor between pins 6 and 7, and C is the value of the capacitor that connects pin 2 to ground. In our circuit, the frequency works out to about 70Hz (a period of about 15mS)--as this was a suggested frequency for switch-debouncing applications on one website we found.

The 74374 IC (also available as 74LS374, 74HC374, etc.) provides eight D-type latches in one package; because we are interested in latching ten digital signals, we need two ICs working in tandem. The inputs are usually referred to as "Dxxx" (data) and the outputs are usually referred to as Qxxx (state). The /OE input, when asserted (active-low), tells the latches to output their current state (if this signal is not asserted, the outputs will "float" so that other components can use the same data bus). As we do not have any other components using this bus, we can tie the /OE pins to ground so that the outputs will always the current state of the latches.

While this debouncing approach works, it may be considered overkill for our needs. Having three extra ICs uses valuable breadboard space.

Another approach to hardware debouncing is to use "debouncing capacitors". This is a much more space-economical solution. See fig. 4:

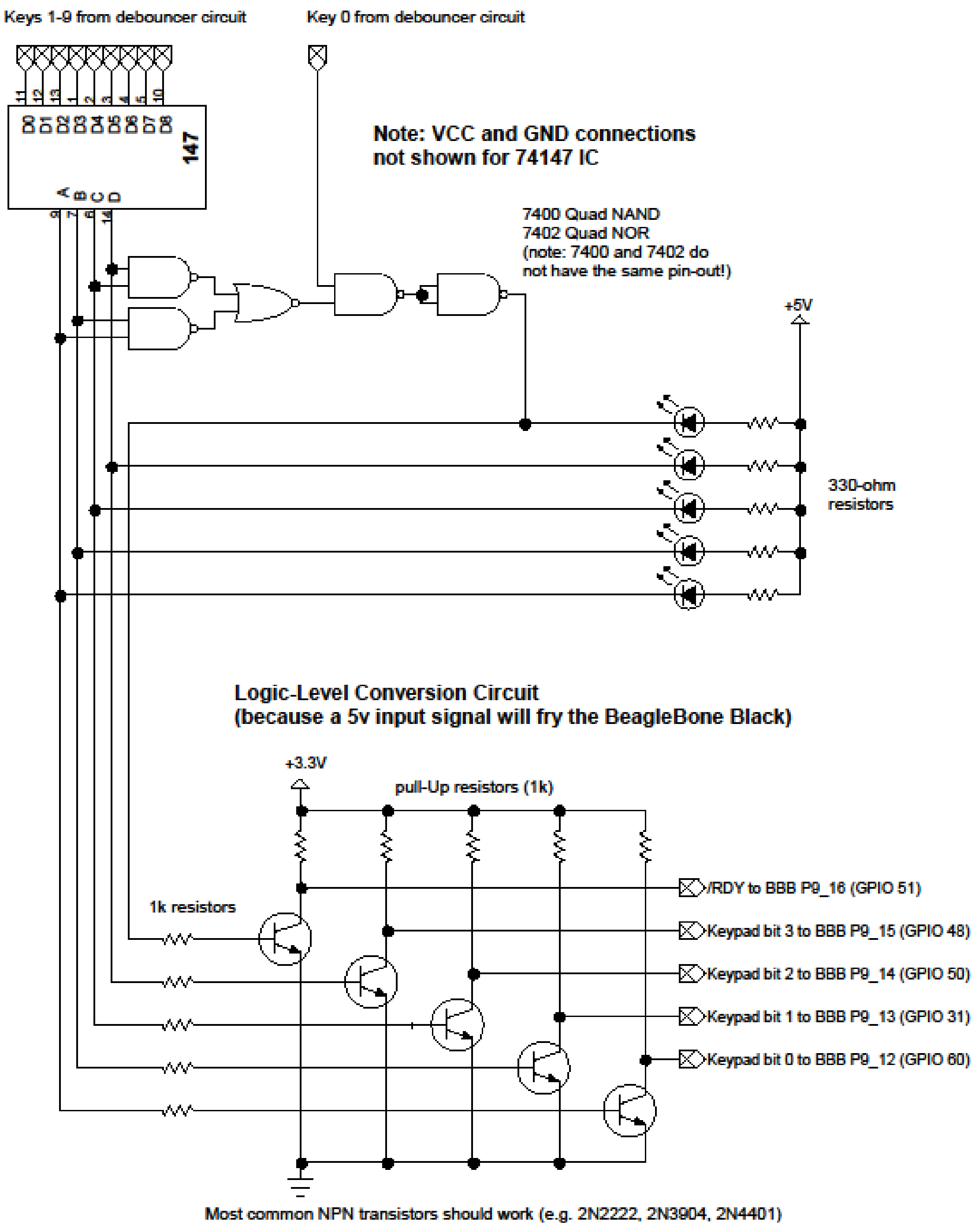
Fig. 4: Keypad Debouncer Circuit using Capacitors



Notice that when a keypad button is pressed, it shorts the corresponding capacitor, causing it to discharge. In our testing, we found this to be an equally-reliable approach to using the clocked latches, and ultimately chose this option.

Now that we've obtained cleaner input signals, we can either use them as-is (thus requiring ten GPIO pins on the BeagleBone Black), or we can encode the keypress into binary to reduce the number of pins down to five (four bits are sufficient for binary/BCD values of 0-9, and a fifth bit indicates that the value is ready to be read [to distinguish the "0" key from a lack of any input]). The 74147 IC (also available as 74LS147, 74HC147, etc.) is a "ten-to-four" priority encoder, which takes the highest-numbered asserted input and encodes that as a four-bit number on its output pins. For whatever reason, the 74147 actually only has *nine* inputs--which is bothersome. We will have to generate the "ready" signal ourselves. See fig. 5 for our implementation.

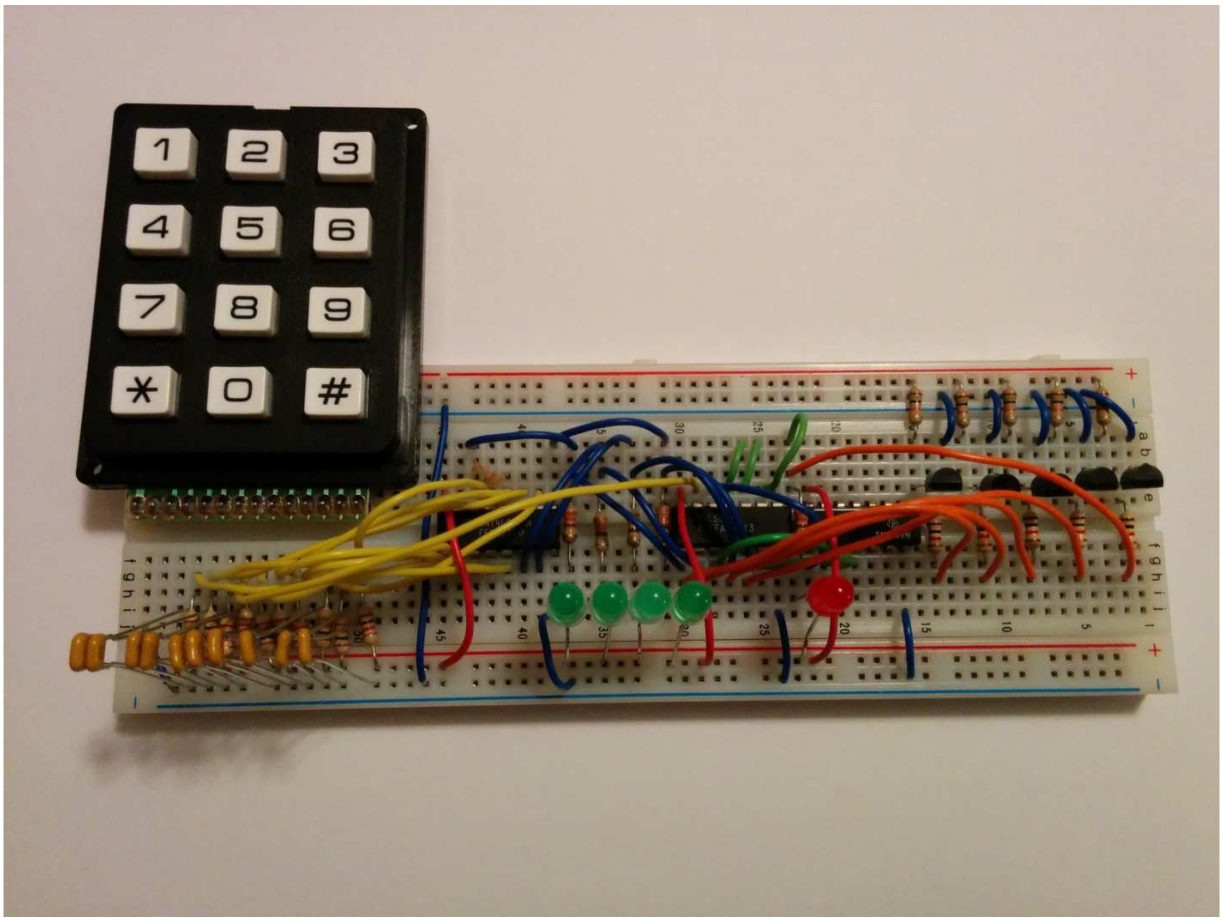
Fig. 5: Keypad Encoder Circuit using 74147, 7400, and 7402


Keys 1-9 from debouncer circuit
Key 0 from debouncer circuit
11 12 13 1 2 3 4 5 10
D0 D1 D2 D3 D4 D5 D6 D7 D8
147
A B C D
9 7 6 14
Note: VCC and GND connections not shown for 74147 IC
7400 Quad NAND
7402 Quad NOR
(note: 7400 and 7402 do not have the same pin-out!)
+5V
330-ohm resistors
Logic-Level Conversion Circuit
(because a 5v input signal will fry the BeagleBone Black)
+3.3V
pull-Up resistors (1k)
1k resistors
/RDY to BBB P9_16 (GPIO 51)
Keypad bit 3 to BBB P9_15 (GPIO 48)
Keypad bit 2 to BBB P9_14 (GPIO 50)
Keypad bit 1 to BBB P9_13 (GPIO 31)
Keypad bit 0 to BBB P9_12 (GPIO 60)
Most common NPN transistors should work (e.g. 2N2222, 2N3904, 2N4401)

The five logic gates amount to a five-input NAND gate, which outputs a 0 when all inputs are 1, and 1 otherwise (when any key is pressed). We attached five LEDs to the four data signals and the ready signal to facilitate debugging, and convince ourselves that the circuit was working properly. Note that the "ready" signal is active-low in this implementation.

The final step in preparing the signals for consumption by the BeagleBone Black is to convert them from 5 volts to 3.3 volts, as this is the maximum voltage the BBB is rated to accept on GPIO input pins. We used five NPN transistors to accomplish this. The BBB provides a 3.3v source that we can use for this sub-circuit. Note that if you are powering the BBB via USB (instead of through the AC adapter), you may only get ~1.8v on the nominal 3.3v pins. Also note that you will need to tie the ground of your 5.5v supply with the ground of your 3.3v supply (or the BBB itself); this isn't shown in our schematic.

Fig. 6: Completed Circuit (5v, 3.3v, ground connections, and data output wires not shown for clarity; we used the top red rail for 3.3v and the bottom red rail for 5v. There is an error in this photo: the keypad should be moved down so it makes electrical contact with the bottom half of the breadboard.)



We arbitrarily chose five GPIO pins that were contiguous on the P9 header, and known to not conflict with anything else, for our inputs.

To configure the GPIO pins as inputs, you will need to write some values to the /sys/class/gpio hierarchy. This can be accomplished either with a shell script, or with C code. Note that if you use a shell script, you will need to call it before the C program runs when the BBB boots (you could call it from /home/root/.profile). Our script is as follows:

```
# gpio-init.sh

# export P9 header GPIO pins 12, 13, 14, 15, 16

# keypad bit 0 -> pin 12 --> gpio60
# keypad bit 1 -> pin 13 --> gpio31
# keypad bit 2 -> pin 14 --> gpio50
# keypad bit 3 -> pin 15 --> gpio48
# keypad ready -> pin 16 --> gpio51

echo 60 > /sys/class/gpio/export
echo 31 > /sys/class/gpio/export
echo 50 > /sys/class/gpio/export
echo 48 > /sys/class/gpio/export
echo 51 > /sys/class/gpio/export

echo in > /sys/class/gpio/gpio60/direction
echo in > /sys/class/gpio/gpio31/direction
echo in > /sys/class/gpio/gpio50/direction
echo in > /sys/class/gpio/gpio48/direction
echo in > /sys/class/gpio/gpio51/direction
```

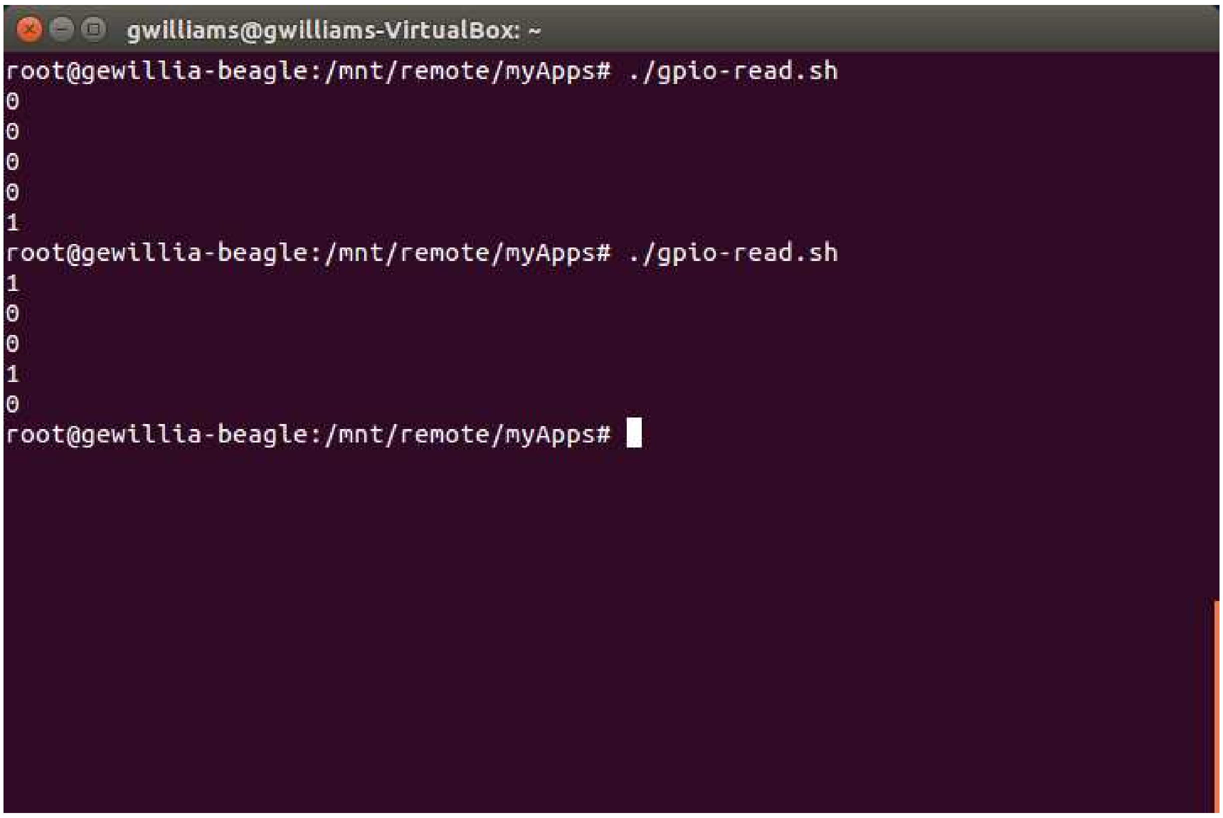
At this point you should be able to test that the BBB is capable of reading keypad values by running a second shell script:

```
# gpio-test.sh

cat /sys/class/gpio/gpio60/value
cat /sys/class/gpio/gpio31/value
cat /sys/class/gpio/gpio50/value
cat /sys/class/gpio/gpio48/value
cat /sys/class/gpio/gpio51/value
```

This will dump the status of the five GPIO lines to the console.

Fig. 7: Output of gpio-test.sh with no input, and while the "9" key is being pressed. From top-to-bottom, the outputs are bit 0, bit 1, bit 2, bit 3, and /ready (it is active-low).



```
root@gewillia-beagle:/mnt/remote/myApps# ./gpio-read.sh
0
0
0
0
1
root@gewillia-beagle:/mnt/remote/myApps# ./gpio-read.sh
1
0
0
1
0
root@gewillia-beagle:/mnt/remote/myApps#
```

It should be possible to configure the "ready" input as an interrupt trigger, but for our purposes this was not necessary, so we leave the implementation as an exercise for the reader. Instead, our C program (which runs in user space) runs a thread that polls the "ready" signal every 50mS. Here is some sample code to get you started.

First, we define some constants, including the paths to the GPIO value files of interest:

```
static const int const NUM_BITS = 4;

static const char const NOT_READY = '1';
static const char const READY = '0';

static const char * const KEYPAD_BIT_FILE_NAMES[] = {
        "/sys/class/gpio/gpio60/value",
        "/sys/class/gpio/gpio31/value",
        "/sys/class/gpio/gpio50/value",
        "/sys/class/gpio/gpio48/value",
};

static const char * const KEYPAD_READY_FILE_NAME =
"/sys/class/gpio/gpio51/value";
```

We also define a wrapper for nanosleep() to wait the specified number of milliseconds:

```
/*
        Waits the specified number of milliseconds.
*/
static void sleep_ms(int ms)
{
        long seconds = 0;
        long nanoseconds = 1000000 * ms;
        struct timespec reqDelay = {seconds, nanoseconds};
        nanosleep(&reqDelay, (struct timespec *) NULL);
}
```

Some utility functions to wait for key events:

```
/*
        Waits until the 'keypad ready' signal is in the desired state.
*/
static void wait_for_key_event(char desired_state) {
        FILE *file = fopen(KEYPAD_READY_FILE_NAME, "r");

        while(1) {
                fseek(file, 0, SEEK_SET);
                char buffer[10];
                fread(buffer, sizeof(char), sizeof(buffer) - 1, file);

                printf("\n"); // if this line is commented out, the program
will stop working--we did not have time to determine why this was needed

                if (buffer[0] == desired_state) {
                        fclose(file);
                        return;
                }

                sleep_ms(50);
        }
}


/*

        Waits for the user to press a key on the keypad.
*/
static void wait_until_ready() {
        wait_for_key_event(READY);
}

/*
        Waits for the user to release all the keys on the keypad.
*/
static void wait_until_not_ready() {
        wait_for_key_event(NOT_READY);
}
```

In practice, the buffer will only ever have two characters in it, so it's not necessary to allocate a 10-character array. We found it necessary to fseek() to the beginning of the file in each iteration of the loop, though this seems redundant.

And finally, the function that translates the input signals into a usable value:

```
/*
        Returns the ASCII character ('0'->'9') corresponding to the next
keypress on the keypad.
*/
static char read_keypad() {
        wait_until_not_ready();
        wait_until_ready();

        int result = 0;

        for (int i = 0; i < NUM_BITS; i++) {
                FILE *file = fopen(KEYPAD_BIT_FILE_NAMES[i], "r");
                char buffer[10];
                fread(buffer, sizeof(char), sizeof(buffer) - 1, file);

                if (buffer[0] == '1') {
                        result |= 1 << i;
                }

                fclose(file);
        }

        return '0' + result;
}
```

What you do with the resulting keypress value is, of course, up to you. From our keypad-scanning thread, we chose to pipe the key character back to the main thread so that it could be displayed in stdout, and also wrote the value to a socket that our webserver was listening to.

We found that our C code worked reliably with the 50mS wait period on the first attempt, and did have a need to try other polling frequencies. Experiment with this value as needed.

We hope this guide is useful to you!

We hereby grant permission to Dr. Brian Fraser to make this guide freely available to other students.