Hex Keypad Explanation

Introduction

The hex keypad is a peripheral that connects to the DE2 through JP1 or JP2 via a 40-pin ribbon cable. It has 16 buttons in a 4 by 4 grid, labelled with the hexadecimal digits 0 to F. An example of this can been seen in Figure 1, below.

Internally, the structure of the hex keypad is very simple. Wires run in vertical columns (we call them C0 to C3) and in horizontal rows (called R0 to R3). These 8 wires are available externally, and will be connected to the lower 8 bits of the port. Each key on the keypad is essentially a switch that connects a row wire to a column wire. When a key is pressed, it makes an electrical connection between the row and column. The internal structure of the hex keypad is shown in Figure 2. The specific mapping of hex keypad wires (C0 to C3 and R0 to R3) to pins is given in Table 1.

C	D	E	F
8	9	A	B
4	5	6	7
0	1	2	3

Figure 1: hex keypad layout

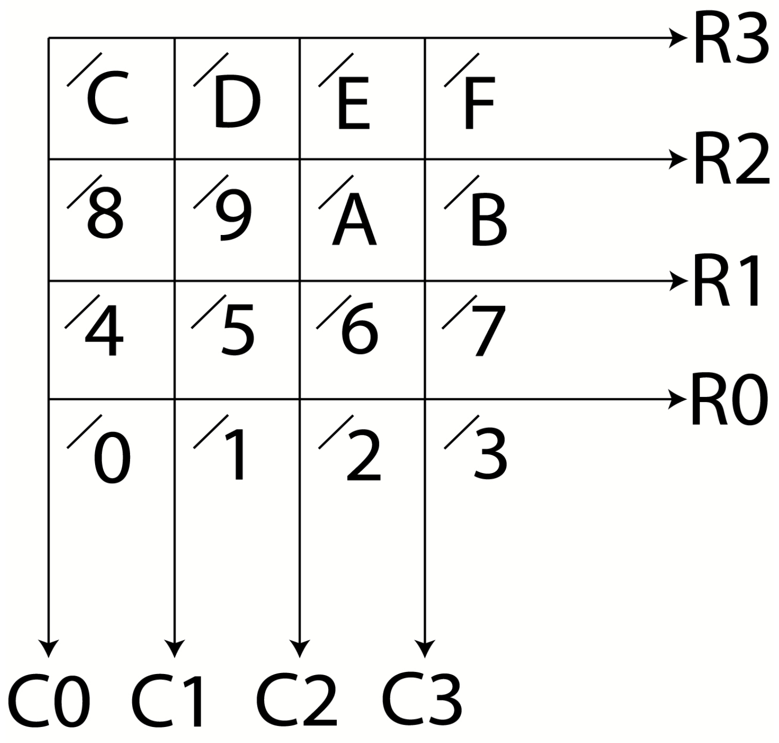

Figure 2: hex keypad internal wiring

Table 1		
Hex Keypad pins	**JP1 pin(s)**	**JP2 pin(s)**
R0	0	0
R1	1	1
R2	2	2
R3	3	3
C0	4	4
C1	5	5
C2	6	6
C3	7	7

At this point, you may be wondering exactly where the signals on the hex keypad come from. The keys just create a short between a row and column wire when pressed, but the row and column wires all come from JP1 or JP2, rather than connecting to power or ground.

Reading Values from the Hex Keypad

It is tempting to view the hex keypad as a peripheral which just tells us which key was pressed, and all we have to do is read the value via the GPIO port. This is the wrong view to take. The hex keypad is just a way for a user to interact with the DE2 board. As described in the previous section, all the keypad does is make electrical connections between rows and columns – it is up to your program to determine from that which key was pressed.

The hex keypad is connected to the DE2 Media Computer via the GPIO parallel ports. We need to remember a few things about the GPIO ports in order to read and interpret hex keypad input properly.

Recall that each pin on JP1 or JP2 can be configured individually as input or output. Furthermore, the port direction can be reconfigured by your program, so that the inputs and outputs can be changed while your program is running. Finally, remember that each pin in the HEX keypad is connected to a pull-up resistor, so any input coming from the hex keypad will read a 1 by default (i.e. when a key is not being pressed).

These facts, coupled with our knowledge of how the row and column wires of the hex keypad are wired up to the DE2 board, will allow us to determine which key has been pressed.

If we treat all the hex keypad wires as inputs, we will always read in a 0xFF, since there is nothing driving those wires – they are unconnected. Even when a key is pressed, the effect is of connecting one input port to another, so the pull-up resistors will always output a 1.

The basic concept is that, since all the hex keypad wires are connected to JP1/JP2, we need to use some of those wires to output values, and some to read in values. If we output values onto the columns, say, then when we read from the rows, if a key is pressed there will be a short between a row and a column and we will read in whatever value we have set the column to output.

Rows in which no key is pressed will be unconnected, and thus read in as a 1. Consequently, in order to be able to differentiate between unconnected inputs and inputs for which a key has been pressed so that they are reading the value put onto a column, we need to always output 0.

Thus, we should write 0 to all of the column wires, then read in from the row wires. If no key is pressed, the row wires will all be unconnected, so the we will read a 1 value on pins 0 to 3 of JP1/2. However, if a key is pressed, one of the row wires will be shorted with a column wire, and will thus have whatever value is on that wire (i.e. 0). Thus, we can identify the *row* of which key was pressed by reading the row values and finding which is 0.

This could also work if we treated the rows as outputs and the columns as inputs. In that case, we would output 0 to the rows, and read in from the columns, and whichever column wire was 0 would indicate the *column* in which the pressed key resides.

So, being able to identify the row or column of a pressed key helps, but still does not tell us which key was pressed. The trick is that if you know both the row *and* the column, you can determine which key was pressed from their intersection.

This means that we must take advantage of the ability to change the direction of the individual pins of JP1/2 within our program. First, we set one half of the lower 8 bits as input (bits 0 to 3, say) and the other as output (bits 4 to 7), and get one value (the row, in this case), then we set them the other way around, get the other value, and then we determine which key was pressed.

So, for example, if we read in that row wire R2 is 0 and column wire C3 is 0, we know that the B key on the hex keypad was pressed. There are a number of different ways you can write code that will figure out which key was pressed based on the row and column numbers.

Thus, to summarize, the following steps should be followed in order to determine which key on the hex keypad has been pressed.

1. JP1 or JP2 should be configured to have the pins connected to row wires R0 to R3 set as inputs. The pins connected to column wires C0 to C3 should be set as outputs. (That is, pins 0–3 are inputs, and 4–7 are outputs.)

2. The outputs (bits 4–7) should be set to 0.

3. Whichever input (bits 0–3) reads in as 0 indicates the row of the pressed key.

4. JP1/JP2 should then be reconfigured to have the pins connected to row wires R0 to R3 set as outputs, and the pins connected to column wires C0 to C3 should be set as inputs. (That is, bits 0–3 are outputs, and 4–7 are inputs.)

5. The outputs (bits 0–3) should be set to 0.

6. Whichever input (bits 4–7) reads in as 0 indicates the column of the pressed key.

7. Now that the row and column of the pressed key are known, you can use software to determine their intersection and thus which key on the hex keypad was pressed.

Debouncing

Unfortunately, reality now rears its ugly head. While the above process is correct, we have to deal with the fact that the keys are not ideal switches. Thus, while we may represent the keys as an ideal electrical switch connecting a row wire and column wire (as shown in Figure 3), the reality is that the switch is a mechanical device subject to the laws of physics.

Thus, when the switch is closed, as contact is made the connector will "bounce" open and closed for about 10 milliseconds. (You can view it much like a door that rattles in its frame after it has been slammed – the rebound of the physical force closing it causes some bouncing.)

This means that when we try to read the hex keypad, if we read in that 10 millisecond "bounce" window, we may be reading at a time when the switch is open, and thus not get the correct value. The solution to this is to debounce the input.

Debouncing is achieved by simply waiting for the inputs to stabilize before taking a reading. Thus, after detecting a keypress (whether via interrupt or polling), your program should wait for approximately 10 milliseconds before reading the row and column wires as described in the preceding section. A delay of 10 milliseconds can be generated by executing a short loop roughly 600 times.

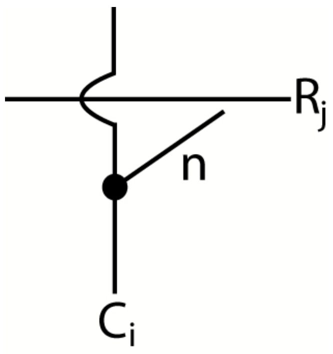

Figure 3: ideal hex keypad switch

Problems You Might Encounter

Because the hex keypad is a mechanical device, it operates much more slowly than your program will run on the microprocessor. If, say, you are using polling to read the hex keypad, it is likely that once you finish reading in a value, the same keypress will register again immediately. In other words, a single keypress will be read multiple times.

As an example, assume you have your program written so that it keeps checking for keypresses and displays the key pressed on the terminal. If you use polling, so that you are simply constantly checking for a keypress, what you will see is that a single press of a key will cause perhaps 20 characters to be displayed by your program. That is because the amount of time the switch remains closed – even with debouncing – is much longer than the amount of time it takes for your program to read the value.

There are a few ways you might deal with this. If you are primarily concerned with determining when a new or different key is pressed, then you might modify your program to compare the current value being read to the last value that was read, and only display a character if the value is different. This would be suitable for programs in which you are maintaining a state, and you only need to know when you have to change that state.

Alternately, you could simply sample the input from the hex keypad at regular intervals. This is more common for programs in which you need to have continual input. Many games would work this way – they would read input from the user, apply changes to the game state, redraw the graphics based on the new state, and repeat.

Putting a lengthy delay at the end of your reading code could also work, but is rather inefficient, since your program will be just wasting cycles. Either of the two preceding options would be preferable.

You should also consider what happens when a key is released while you're reading it, so that you get a valid value for the row, but no value for the columns (or vice versa). Your program should discard such inputs, since you cannot identify the key that was pressed.

Using the Hex Keypad with Interrupts

The hex keypad can be used with interrupts. The only thing that needs to be determined is if you want to use the rows or columns to generate interrupts. For row interrupts, enable interrupts on bits 0–3 and for columns enable interrupts on bits 4–7.

JP1 has IRQ of 11, and JP2 has an IRQ of 12. It is important to keep these differences in mind if you are relying on the priority level of the interrupt for some key functionality.

So if interrupts are being used with the HEX keypad, the DE2 board must be configured to generate them appropriately. A key press on a row or column should trigger an interrupt, and then debouncing should be done before reading the rows or columns. Lastly, before exiting the interrupt routine, column and row wires should be set back to their original state.

The interrupts can be configured as rising-edge triggered or falling-edge triggered. For the most reliable results it is best to generate interrupts on a falling edge, as apposed to a rising edge. This is due to debouncing issues. If the key press sticks a bit you may generate multiple interrupts. It should also be noted that you may want to generate a longer delay to ensure only one interrupt happens per key press.

General Difficulties with Interrupts

Getting your program to work with interrupts can be a tricky task. While this document is not meant to go into detail about how to get interrupts working, there are some general pointers that may help.

First of all, you may notice that interrupts work very erratically as you develop, test, and debug your program. Your program will work on some occasions, and not on others. While there can be many causes for this, it is probably a combination of old or incorrect interrupt configuration still residing in system control registers and uncleared interrupts still registering, meaning programs never quite work right. The best solution if you find erratic behaviour is to power off the DE2 board for at least 10 seconds. This should be long enough to flush all old configurations, so you can try again.

Secondly, make sure you always have a sample program that works. Test it with your DE2 board and make sure it works. Once you are confident that a simple version works, move on to build and test your own.

Document History:

Original version:	Andrew House, March 2006
Revision 2:	Fred Aulich, September 2007
Revision 3:	Andrew House, November 2009