Lynne Lammers
EECS 388
Exercise 3 Questions
Due 2/17/14

1. Port Configuration: [10 points]
   What is a pull-up resistor?
   A pull-up resistor is a resistor connected to a source of high voltage that keeps the voltage of the connected device high ("pulls up" the voltage) when the circuit is open.

   Explain the GPIOPadConfigSet function.
   - What does this function do?
   This function configures the drive strength (for an output pin) and the type of pin, such as weak pull-up or weak pull-down for the specified pin or pins. (Stellaris Peripheral Drive Library, p. 156)
   - What are the formal parameters?
   The formal parameters are ulPort (the base address of the GPIO port), ucPins (the bit-packed pins), ulStrength (the drive strength), and ulPinType (the pin type). (Stellaris Peripheral Drive Library, p. 156)
   What arguments did you use in your speaker and button initializations?
   In the speaker initialization, the ulPort argument was GPIO_PORTH_BASE (port H), and the ucPins argument was GPIO_PIN_1 | GPIO_PIN_0 (the bitwise OR of pins 1 and 0). For the button initialization, the ulPort argument was GPIO_PORTG_BASE (port G), and ucPins argument was ButtonPins, which was defined as GPIO_PIN_3 | GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7 (the bitwise OR of pins 3-7). For both the speaker and button initializations, the strength argument was GPIO_STRENGTH_2MA (2 mA) and the pin-type argument was GPIO_PIN_TYPE_STD_WPD (a push-pull weak pull-down type pin).

2. Pin Configuration: [10 points]
   Explain the GPIOPinTypeGPIOOutput and GPIOPinTypeGPIOInput functions.
   - What does the function do?
   GPIOPinTypeGPIOOutput configures pins to be used as outputs and GPIOPinTypeGPIOInput configures pins to be used as inputs. (Stellaris Peripheral Drive Library, p. 164-5)
   - What are the formal parameters?
   For both functions, the first parameter is ulPort, which is the base address of the GPIO port, and the second parameter is ucPins, which is the bitpacked pin or pins you want to configure.
   What arguments did you use in your speaker and button initializations?
   In the speaker initialization, for the base address of the GPIO port, I used GPIO_PORTH_BASE, and for the bit-packed pins, I used GPIO_PIN_1 | GPIO_PIN_0 (the bitwise OR of pin 1 and pin 0). In the button initialization, for the base address of the GPIO port, I used GPIO_PORTG_BASE, and for the

bit-packed pins, I used ButtonPins, which was defined as GPIO_PIN_3 | GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7 (the bitwise OR of pins 3-7).

3. Switch De-bouncing: [15 points]
   What is switch bounce?

   Switch bounce occurs when the mechanism in the switch bounces on contact, creating a rapid alteration between high and low voltage before the switch settles to its final contact voltage (low in this case).

   Why is de-bouncing a switch signal necessary? Explain your de-bouncing algorithm.
   - How does it work exactly?

   It is important to de-bounce a signal because if you read the signal exactly as-is, it would be interpreted as several individual toggles of the switch, rather than a single push of the switch. To screen out the bounce in this lab, a waiting period is used after initial detection of a button push and then the state of the pins is checked again. This skips over the bounce between low and high signals after a press, and then checks the state again to confirm it remains low. To do this, first the current state of the pins is read into a variable called currState. This is compared to the all-off state of the pins (ButtonPins), and if any pins are different, that signifies that a button press is detected. Next, currState is compared to the variable prevState. If a button had *just* changed state (initial detection of button press), then prevState and currState would differ, so this if-statement would not be carried-out and we would enter the else-statement that implements the waiting-period or buffer to filter out the bounce. The else-statement adds 90 to monitorNext (the next sysTickCount to execute the button checks), and finally, at the end of the execution task, monitorNext is incremented again by 10 (monitorDelta) to add a total of 100, or 10 ms, to monitorNext. The button check is carried out again in 10 ms, after any bounce would have settled, and if the same button is indeed still pressed (determined by comparing currState and prevState), the program can carry out the actions for a button press.

4. Button Press Detection: [15 points]
   The program can check button states several hundred times per second, this means that even very short button presses will register several times.
   How did your program insure that the appropriate action was only executed once, even if the button was held down?

   The program used three variables to keep track of three button states at once. The variables were currState, which holds the state of the pins at the time of execution, and prevState & origState, which hold the two previous states of the pin. By keeping track of the three most recent states, the program could determine whether a button press had just occurred or if a

button was continuing to be depressed, and play the speakers only when the button state changed. Once the program determined that a button had been pressed, before changing the flag for the speakers to sound, two checks occurred. First, if the current state and previous state both had the same button depressed (i.e. if currState = prevState), then the second check was carried out. Next, the current state was compared to the state two executions ago (origState). If these were different, that meant the button has newly been pressed (and confirmed pressed after de-bouncing), and the code to set the speaker flag to play the sound was executed. However, if currState and origState were the same, this meant that the button was just being held down and not newly depressed, so the code to set the speaker flag was not executed.

How did you program detect when the button was released?

Again, the three states were utilized to determine when the button was released. First, the current state (currState) of the pins was compared to the all-off state (ButtonPins). This comparison was done by taking the bitwise AND of currState and ButtonPins and comparing determining if that was equivalent to ButtonPins. If currState and ButtonPins were different, that meant a button was depressed, and therefore a button had not been released. However, if they were the same, a second check was carried out to determine whether a button had just been released or not. In this check, the current state was compared to the previous two states. If the current state and previous state were the same (both with no buttons pressed), but the current state and the state two executions ago were different (i.e. origState had a button depressed), then a button had just been released, and the code for the speaker flag and UART display was executed. However, if in all three states, no buttons were pushed (i.e. currState = prevState and currState = origState), then no button had just been released and the code for a release was not executed.

How did you detect which specific button was pressed?

Once a release had been detected, I had to determine which button had been released in order to display it. Since origState still held the pin data for the pressed button, I used that variable in my determination. I used several if-statements to compare a bitwise XOR of the value in origState and each of the button pin values (stored in UpButton, DownButton, LeftButton, RightButton, and SelectButton) with ButtonPins. If the two were equilavent, then I had the correct button and could send a message with UART. For example, if origState held the value 11101000, a bitwise XOR with the value in UpButton, pin 3 (00001000) would produce 1110000, which would not be equivalent to the value in ButtonPins (11111000). However, a bitwise XOR of origState and DownButton, pin 4 (00010000) would produce 11111000, which would be equivalent to ButtonPins. Therefore DownButton was the button that was released, and the code to display that DownButton had been released was carried out.

5. Inter-process communication: [10 points]
   How does your button monitor task signal the speaker task that it is time to emit sound? How does the program control the length of the tone emitted?

   In order to signal the speaker task to emit a sound, a global flag variable called speakerFlag of type short was used. The code inside SpeakerBuzzExecute that executes the buzz was placed inside an if-statement that checked the value of speakerFlag. If speakerFlag equaled 1, the SpeakerBuzzExecute code could be carried out. However, if speakerFlag equaled anything else, it wouldn't be carried out and no tone would play. In the ButtonExecute code, anytime a button was detected to be pressed or released, speakerFlag was set to 1, allowing SpeakerBuzzExecute to carry out the code to emit a tone.

   There were two components to control the length of the tone. The first was that at the beginning of the ButtonExecute code, inside the if-statement that checks whether it's time to execute again by looking at monitorNext, speakerFlag was set to 0. Therefore, anytime the ButtonExecute code was carried out, the tone would stop emitting. The other component was to alter the time that the ButtonExecute code would be carried out. This was accomplished by incrementing monitorNext. For example, if a button press was detected, speakerFlag would get set to 1 to allow the tone to play, then 1990 would be added to monitorNext. At the end of the ButtonExecute code, monitorNext would be updated by monitorDelta (10) as well, meaning it was incremented by a total of 2000, or .2 s. Therefore, the ButtonExecute code would not be carried out until 2000 sysTicks had passed, and speakerFlag would keep a value of 1 that entire time, producing a tone for .2 seconds. Similarly, if a button release was detected, monitorNext was incremented by 4990 at detection, and again by 10 at the end of the code. This allowed 5000 sysTicks, or .5 seconds to pass before executing the ButtonExecute code again and resetting speakerFlag to 0.