```
//	Evan Heinrich
//	CE2801 sect. 011
//	10/19/2021
//
//	File:
//		main.S
//	Description of File:
//		Driver program for lab 6
//		Supposed to be similar to a safe keypad,
//		user enters a pin and presses an enter button.
//		Will emit a success tone and light an LED, where
//		the LED is an example of activating a servo or something
//		if the pin was correct, otherwise a failure tone emits.
//	(opt) Dependencies:
//		timer_delay.S
//		LCD_Control.S
//		keypad.S
//		ASCII.S
//		LED.S
//		tone.S


// Program flow
// Get character of keypad (blocking)
// Compare to '*' and '#'
//		If '*' clear all entries
//		If '#' compare string to code
//		If neither, push character to display
// If a char was displayed, increment counter for number of characters
// If number of chars displayed = 16, newline
// If number of chaes displayed = 32, attempt code
// If string = code
//		Light an LED
//		Play success tone
//		Only reset by reset button
// If string != code
//		Play fail tone
//		Only reset by reset button

.syntax unified
.cpu cortex-m4
.thumb
.section .text

.equ STRING_MAX, 32
.equ LINE_MAX, 16

.global main
```

```
main:
	// Initialization
	BL delay_Init
	BL LCD_Init
	BL Key_Init
	BL LED_Init
	BL Tone_Setup

	// Startup Messages
	BL startup

	// String size iterator
	MOV R7, #0

	// Line position
	MOV R6, #0

loop:
	BL Key_GetChar      // Get character that was pressed
	MOV R1, R0          // Move into argument register

	CMP R1, #0x2A       // Compare to the char 0x2A, AKA '*'
	IT EQ
		BEQ clear    // If the character was '*', clear entry

	CMP R1, #0x23       // Compare to the char 0x23, AKA '#'
	BEQ compare         // If the character was '#', compare to the actual code

	CMP R7, #LINE_MAX   // Compare string length to display width
	IT EQ
		BLEQ newline // If there are 16 chars on display, start a new line

	CMP R7, #STRING_MAX // Compare string to max width
	BEQ compare         // If the string is at max size, compare to the actual code

	// If none of the above comparisons trigger, print the char and update entry
	BL LCD_PrintChar    // Print the character
	LDR R0, =ENTRY      // Load entry address
	STRB R1, [R0, R7]   // Store the char at the current index
	ADD R7, #1          // Increment string size

	B loop
```

```
compare:
	// Use my string comparison method in ASCII.S to
	// compare the user entry to the actual code
	LDR R1, =CODE			// Load the address for the actual code
	LDR R2, =ENTRY			// Load the address for the user entry
	BL ASCII_StringCompare	// Compare the two

	MOV R2, #0				// Clear success/fail light

	CMP R0, #0				// Compare the results of string comparison to 0, aka equal
	ITTT EQ
		LDREQ R1, =good		// If they were equal, load the success message
		MOVEQ R2, #1		// If equal, prep success LED
		BLEQ Tone_Success	// Play the success tone

	CMP R0, #0				// Playing success tone updates CPSR so redo comparison

	ITT NE
		LDRNE R1, =fail		// If they weren't equal, load the fail message
		BLNE Tone_Failure	// Play the failure tone

	BL LCD_Clear
	BL LCD_PrintString		// Print the success/fail message
	MOV R1, R2				// Prep to display success/fail LED
	BL num_to_LED			// Display success/fail LED

	// 3 second delay
	MOV R1, #3
	BL delay_sec

	// Clear entry
	B clear


newline:
	PUSH {R0, R1, LR}

	CMP R6, #0				// Determine which line we are on
	ITTTT EQ				// If on first line
		MOVEQ R0, #1		// Second line
		MOVEQ R1, #0		// First column
		MOVEQ R6, #1		// Update line counter
		BLEQ LCD_MoveCursor // Move cursor

	POP {R0, R1, LR}
```

```
clear:
	PUSH {R0, R1, R2}

	MOV R0, #0			// Iterator
	MOV R2, #0			// Clear value
	LDR R1, =ENTRY			// Entry address

	1:
	STRB R2, [R1, R0]	// Overwrite 0
	ADD R0, #1		// Increment iterator
	SUBS R7, #1		// Decrement string size
	BNE 1b			// Loop until string size is negative
	MOV R7, #0		// Clear string size
	BL LCD_Clear		// Clear display

	POP {R0, R1, R2}
	B loop			// Return to loop

startup:
	PUSH {R0-R1, LR}

	// First line of first message
	LDR R1, =msg1_1
	BL LCD_PrintString

	// Move to second line
	MOV R0, #1
	MOV R1, #0
	BL LCD_MoveCursor

	// Second line of first message
	LDR R1, =msg1_2
	BL LCD_PrintString

	// Wait 5 seconds
	MOV R1, #5
	BL delay_sec

	// Clear display
	BL LCD_Clear

	// First line of second message
	LDR R1, =msg2_1
	BL LCD_PrintString

	// Move to second line
	MOV R0, #1
	MOV R1, #0
	BL LCD_MoveCursor

	// Second line of second message
	LDR R1, =msg2_2
	BL LCD_PrintString

	// 5 second delay
	MOV R1, #5
	BL delay_sec
```

```
	// Clear display
	BL LCD_Clear

	// Run rest of program
	POP {R0-R1, PC}


.section .rodata
CODE:  .asciz "71293"
good:  .asciz "Success!"
fail:  .asciz "Wrong code!"
msg1_1:      .asciz "Push buttons on"
msg1_2:      .asciz "the keypad"
msg2_1:      .asciz "Press * to clear"
msg2_2:      .asciz "Press # to enter"

.section .data
ENTRY: .byte 0
```

```
//     Evan Heinrich
//     CE2801 sect. 011
//     9/28/2021
//
//     File:
//           tone.S
//     Description of File:
//           Initially will just hold an example success and failure tone
//           and the code to make the MSOE development board piezo buzzer
//           emit those tones
//     (opt) Dependencies:
//           timer_delay.S

.syntax unified
.cpu cortex-m4
.thumb
.section .text

.global Tone_Setup
.global Tone_Success
.global Tone_Failure

// Base addresses
.equ RCC_BASE,      0x40023800
.equ GPIOB_BASE,    0x40020400
.equ TIM3_BASE,     0x40000400

// Offsets
.equ AHB1ENR,       0x30            // AHB1ENR used to enable GPIO ports
.equ APB1ENR,       0x40            // APB1ENR used to enable timers
.equ GPIO_MODER,    0               // Offset from GPIOx base addr to mode register
.equ GPIO_AFRL,     0x20            // Offset from GPIOx base to alt. funct. register (low)
.equ TIM_CR1,       0x00            // Offset from TIMx base to control reg. 1
.equ TIM_ARR,       0x2C            // Offset from TIMx base to auto reload register
.equ TIM_PSC,       0x28            // Offset from TIMx base to prescale register
.equ TIM_CCMR1,     0x18            // Offset from TIMx base to capture/compare mode reg
.equ TIM_CCR,       0x34            // Offset from TIMx base to capture/compare register
.equ TIM_CCER,      0x20            // Offset from TIMx base to capture compare enable reg

// Masks
.equ GPIOBEN,       1 << 1          // Location of the GPIOB enable bit
.equ TIM3EN,        1 << 1          // Location of the TIM3 enable bit
.equ GPIO_ALTFUN,   0b10            // Mask to set a GPIO pin as alternate function
.equ PB4_ALTFUN,    0b0010          // Mask for AFRL to set PB4 as TIM3_CH1
.equ PIN_TOGGLE,    0b011           // Mask to set pin output mode to toggle

// Constants
.equ PRESCALE,      16              // Used to prescale clock from 16MHz to 1MHz
.equ NOTE_C5,       1911            // Pulses of a 1MHz clock to make a C5 note
.equ NOTE_G5,       1276            // Pulses of a 1MHz clock to make a G6 note
.equ NOTE_LEN,      200             // Duration of each note, MS
```

```
//     Function: Tone_Setup
//     Register-safe! Pushes used registers to stack
//     Description:
//           Configures GPIO and timers for use with the MSOE devboard piezo buzzer
//           ->     Piezo buzzer lives on PB4, and one of PB4's alternate functions is
//                  TIM3_CH1
//     Args:
//           N/A
//     Returns:
//           N/A
//     Register Use:
//           R0     -      Scratch
//           R1     -      Addresses
//           R2     -      Scratch
//           R3     -      Scratch
Tone_Setup:
       PUSH {R0-R3, LR}

       // Enable GPIOB
       LDR R1, =RCC_BASE          // Load RCC base addr
       LDR R2, [R1, #AHB1ENR]     // Read from the AHB1ENR
       ORR R2, #GPIOBEN           // Apply GPIOB enable mask
       STR R2, [R1, #AHB1ENR]     // Write back to AHB1ENR

       // Enable TIM3 (enabler also lives in RCC)
       LDR R2, [R1, #APB1ENR]     // Read from the APB1 enable register
       ORR R2, #TIM3EN                   // Apply timer 3 enable mask
       STR R2, [R1, #APB1ENR]     // Write back to APB1ENR

       // Set PB4 as alternate funct
       LDR R1, =GPIOB_BASE                      // Load GPIOB base address
       LDR R2, [R1, #GPIO_MODER]        // Read the mode register
       ORR R2, #GPIO_ALTFUN << (4 * 2)  // Apply the 2 bit mask to PB4
       STR R2, [R1, #GPIO_MODER]        // Write

       // Set alternate function register for PB4
       // PB4 AFR is AFRL [19..16]
       // TIM3_CH1 is alternate function 2
       LDR R2, [R1, #GPIO_AFRL]  // Read current AFRL
       MOV R0, #PB4_ALTFUN              // Load mask for BFI
       BFI R2, R0, #16, #4              // Insert the alt. funct. code into AFRL4
       STR R2, [R1, #GPIO_AFRL]  // Write

       // Update timer prescaler
       LDR R1, =TIM3_BASE
       MOV R2, #PRESCALE
       STR R2, [R1, #TIM_PSC]

       // Configure capture/compare mode register (CCMR)
       // Set output mode to toggle
       // Disable preload
       LDR R1, =TIM3_BASE               // Load Timer 3 base addr
       LDR R2, [R1, #TIM_CCMR1]  // Read from the CCMR
       MOV R3, #PIN_TOGGLE              // Load toggle output mode
       BFI R2, R3, #4, #3               // Insert toggle command
       BFC    R2, #3, #1                       // Clear (disable) preload
       STR R2, [R1, #TIM_CCMR1]
```

```
    // Set CC1E (capture compare ch1 enable)
    // Set CC1P (capture compare ch1 polarity)
    LDR R2, [R1, #TIM_CCER]    // Read
    ORR R2, #0b11 << 0         // CC1E & CC1P live at CCER[1..0]
    STR R2, [R1, #TIM_CCER]    // Write

    POP {R0-R3, PC}
```

```
// Frequencies used (assuming A4 = 440Hz)
//     1. C5 (523.25Hz)
//     2. G5 (783.99Hz)
//
//     C5 -> G5 for success
//     G5 -> C5 for failure
```

```
//	Function: Tone_Success
//	Register-safe! Pushes used registers to stack
//	Description:
//		Uses TIM3_CH1 to play a success tone on the piezo buzzer
//		C5 -> G5
//	Args:
//		N/A
//	Returns:
//		N/A
//	Register Use:
//		R1	-	Delay arguments
//		R2	-	Scratch
//		R3	-	Address
Tone_Success:
	PUSH {R1-R3, LR}

	// Load base address
	LDR R3, =TIM3_BASE

	// Write first frequency
	MOV R2, #NOTE_C5
	STR R2, [R3, #TIM_ARR]
	STR R2, [R3, #TIM_CCR]

	// Turn on clock
	LDR R2, [R3, #TIM_CR1]
	ORR R2, #1
	STR R2, [R3, #TIM_CR1]

	// Play note for the desired length
	MOV R1, #NOTE_LEN
	BL delay_ms

	// Turn off clock
	LDR R2, [R3, #TIM_CR1]
	BFC R2, #0, #1
	STR R2, [R3, #TIM_CR1]

	// Write second frequency
	MOV R2, #NOTE_G5
	STR R2, [R3, #TIM_ARR]
	STR R2, [R3, #TIM_CCR]

	// Turn on clock
	LDR R2, [R3, #TIM_CR1]
	ORR R2, #1
	STR R2, [R3, #TIM_CR1]

	// Play note for the desired length
	MOV R1, #NOTE_LEN
	BL delay_ms

	// Turn off clock
	LDR R2, [R3, #TIM_CR1]
	BFC R2, #0, #1
	STR R2, [R3, #TIM_CR1]

	POP {R1-R3, PC}
```

```
//	Function: Tone_Failure
//	Register-safe! Pushes used registers to stack
//	Description:
//		Uses TIM3_CH1 to play a failure tone on the piezo buzzer
//		G5 -> C5
//	Args:
//		N/A
//	Returns:
//		N/A
//	Register Use:
//		R1	-	Delay arguments
//		R2	-	Scratch
//		R3	-	Address
Tone_Failure:
	PUSH {R1-R3, LR}

	// Load base address
	LDR R3, =TIM3_BASE

	// Write first frequency
	MOV R2, #NOTE_G5
	STR R2, [R3, #TIM_ARR]
	STR R2, [R3, #TIM_CCR]

	// Turn on clock
	LDR R2, [R3, #TIM_CR1]
	ORR R2, #1
	STR R2, [R3, #TIM_CR1]

	// Play note for the desired length
	MOV R1, #NOTE_LEN
	BL delay_ms

	// Turn off clock
	LDR R2, [R3, #TIM_CR1]
	BFC R2, #0, #1
	STR R2, [R3, #TIM_CR1]

	// Write second frequency
	MOV R2, #NOTE_C5
	STR R2, [R3, #TIM_ARR]
	STR R2, [R3, #TIM_CCR]

	// Turn on clock
	LDR R2, [R3, #TIM_CR1]
	ORR R2, #1
	STR R2, [R3, #TIM_CR1]

	// Play note for the desired length
	MOV R1, #NOTE_LEN
	BL delay_ms

	// Turn off clock
	LDR R2, [R3, #TIM_CR1]
	BFC R2, #0, #1
	STR R2, [R3, #TIM_CR1]

	POP {R1-R3, PC}
```

```
//	Evan Heinrich
//	CE2801 sect. 011
//	9/28/2021
//
//	File:
//		timer_delay.S
//	Description of File:
//		Originally created 9/28/2021 for Lab 3
//		Modified 10/19/2021 for Lab 6, conversion to using
//		our board's dedicated timers
//	(opt) Dependancies:
//		N/A

// Assembler Directives
.syntax unified
.cpu cortex-m4
.thumb
.section .text

// Literal Pool
.equ TIM2_BASE,		0x40000000	// Timer 2 base address
.equ RCC_BASE,		0x40023800	// RCC base address

.equ APB1ENR,	0x40		// Offset from RCC base to APB1ENR
.equ TIM_CR1,	0x00		// Offset from TIMx base to control reg. 1
.equ TIM_ARR,	0x2C		// Offset from TIMx base to auto reload register
.equ TIM_PSC,	0x28		// Offset from TIMx base to prescale register
.equ TIM_CNT,	0x24		// Offset from TIMx base to count register

.equ TIM2EN,	1 << 0		// Location of TIM2 enabler is bit 0
.equ OPM_SET,	1 << 3		// Mask to set one pulse mode (do not repeat)
.equ CLK_DIV,	16			// Mask to set clock division to 1MHz
.equ CNT_DN,	1 << 4		// Mask to set count down mode
.equ CNTEN_MASK,	1 << 0		// Mask for the location of counter enable
.equ CNT_MS,	1000		// 1k counts per millisecond w/ 1MHz count rate
.equ CNT_US,	1			// 1 count per microsecond w/ 1MHz count rate
.equ CNT_S,		1000000			// 1M counts per second w/ 1MHz count rate


// Globally exposed functions
.global delay_Init
.global delay_ms
.global delay_us
.global delay_sec
```

```
//     Function: delay_setup
//     Register-safe! Pushes all used registers to the stack
//     Description:
//           Configures TIM2 as a simple countdown timer.
//           ->     Uses a 16x clock division, making the count rate 1MHz
//           ->     TIM2 is a 32-bit counter, allowing for a large range of time
//     Args:
//           Void
//     Returns:
//           Void
//     Register Usage:
//           R1     -      Addresses
//           R2     -      Scratch
delay_Init:
      PUSH {R1-R2, LR}

      // Enable TIM2
      LDR R1, =RCC_BASE          // Load RCC base address
      LDR R2, [R1, #APB1ENR]     // Read
      ORR R2, #TIM2EN            // Apply Timer 2 enable mask
      STR R2, [R1, #APB1ENR]     // Write

      // Set timer configurations
      LDR R1, =TIM2_BASE         // Load Timer 2 base address
      LDR R2, [R1, #TIM_CR1]     // Read
      ORR R2, #OPM_SET           // Apply one pulse mode config
      ORR R2, #CNT_DN            // Apply countdown config
      STR R2, [R1, #TIM_CR1]     // Write

      // Set prescaler
      MOV R2, #CLK_DIV           // Load desired clock division
      STR R2, [R1, #TIM_PSC]     // Apply desired clock division

    POP {R1-R2, PC}
```

```
//	Function: delay_ms
//	Register-safe! Pushes all used registers to the stack
//	Description:
//		Starts a timer for a duration provided in the argument
//		->	Conversion factor is 1,000 so the max value here is
//			4,294,967 and some change.
//	Args:
//		R1	-	Desired timer duration in milliseconds
//	Returns:
//		Void
//	Register Usage:
//		R0	-	Total counts for provided delay
//		R1	-	Argument and Addresses
//		R2	-	Scratch
delay_ms:
	PUSH {R0-R2, LR}

	// Convert the argument in milliseconds to counts
	LDR R2, =CNT_MS			// Load the conversion factor
	MUL R0, R1, R2			// Convert milliseconds to counts

	// Store desired count
	LDR R1, =TIM2_BASE		// Load timer base address
	STR R0, [R1, #TIM_CNT]	// Overwrite counter

	// Start count
	LDR R2, [R1, #TIM_CR1]	// Load the current control register
	ORR R2, #CNTEN_MASK		// Apply mask to enable counter
	STR R2, [R1, #TIM_CR1]	// Write and start count

	// Poll counter until count expires (counter enable = 0)
	1:
	LDR R2, [R1, #TIM_CR1]	// Read control register
	BFC R2, #1, #31				// Clear everything except CEN bit
	CMP R2, #0					// Compare to 0, aka counter expired
	BNE 1b						// Loop if not zero

	POP {R0-R2, PC}
```

```
//     Function: delay_us
//     Register-safe! Pushes all used registers to the stack
//     Description:
//           Starts a timer for a duration provided in the argument
//           ->     Conversion factor is 1 so the max value here is
//                  4,294,967,295
//     Args:
//           R1     -      Desired timer duration in microseconds
//     Returns:
//           Void
//     Register Usage:
//           R0     -      Total counts for provided delay
//           R1     -      Argument and Addresses
//           R2     -      Scratch
delay_us:
      PUSH {R0-R2, LR}

      // Convert the argument in microseconds to counts
      LDR R2, =CNT_US           // Load the conversion factor (technically 1x but still)
      MUL R0, R1, R2            // Convert microseconds to counts

      // Store desired count
      LDR R1, =TIM2_BASE        // Load timer base address
      STR R0, [R1, #TIM_CNT]    // Overwrite counter

      // Start count
      LDR R2, [R1, #TIM_CR1]    // Load the current control register
      ORR R2, #CNTEN_MASK       // Apply mask to enable counter
      STR R2, [R1, #TIM_CR1]    // Write and start count

      // Poll counter until count expires (counter enable = 0)
      1:
      LDR R2, [R1, #TIM_CR1]    // Read control register
      BFC R2, #1, #31                 // Clear everything except CEN bit
      CMP R2, #0                      // Compare to 0, aka counter expired
      BNE 1b                          // Loop if not zero

      POP {R0-R2, PC}
```

```
//     Function: delay_sec
//     Register-safe! Pushes all used registers to the stack
//     Description:
//            Starts a timer for a duration provided in the argument
//            ->     Conversion factor is 1M so the max value here is
//                   4,294 and some change.
//     Args:
//            R1     -      Desired timer duration in seconds
//     Returns:
//            Void
//     Register Usage:
//            R0     -      Total counts for provided delay
//            R1     -      Argument and Addresses
//            R2     -      Scratch
delay_sec:
       PUSH {R0-R2, LR}

       // Convert the argument in seconds to counts
       LDR R2, =CNT_S             // Load the conversion factor
       MUL R0, R1, R2             // Convert seconds to counts

       // Store desired count
       LDR R1, =TIM2_BASE         // Load timer base address
       STR R0, [R1, #TIM_CNT]     // Overwrite counter

       // Start count
       LDR R2, [R1, #TIM_CR1]     // Load the current control register
       ORR R2, #CNTEN_MASK        // Apply mask to enable counter
       STR R2, [R1, #TIM_CR1]     // Write and start count

       // Poll counter until count expires (counter enable = 0)
       1:
       LDR R2, [R1, #TIM_CR1]     // Read control register
       BFC R2, #1, #31                   // Clear everything except CEN bit
       CMP R2, #0                        // Compare to 0, aka counter expired
       BNE 1b                            // Loop if not zero

       POP {R0-R2, PC}
```

Small Addition to ASCII.S

```
//	Function: ASCII_StringCompare
//	Register-safe!
//	Description:
//		Compares the contents of two null-terminated strings to determine if they
//		are equal.
//		The arguments are memory locations to null-terminated strings.
//	Args:
//		R1	-	First String
//		R2	-	Second string
//	Returns:
//		R0	-	0 if equal
//	Register Use:
//		R0	-	Return, 0 if equal
//		R1	-	String1 addr & String1 length
//		R2	-	String2 addr & String2 length
//		R3	-	Iterator
//		R5	-	Backup of String1 length
//		R6	-	Backup of String1 addr
//		R7	-	Backup of String2 addr
ASCII_StringCompare:
	PUSH {R1-R3, R5-R7, LR}

	// Backup addresses
	MOV R6, R1
	MOV R7, R2

	// Get lengths of strings
	BL ASCII_StringLength		// Length of first string
	MOV R5, R0			// Move length into a temp register
	MOV R1, R2			// Move second string into arg register
	BL ASCII_StringLength		// Length of second string
	MOV R2, R0			// Move length into R2
	MOV R1, R5			// Move backup of string 1 length into R1

	// Compare lengths of strings
	CMP R1, R2			// Compare lengths
	ITT NE				// If R1 != R2
		MOVNE R0, #1		// Load 1 into the return register, aka not equal
		BGT return		// Return

	// Now the difficult part. At this point, the strings are the same
	// length, so we need to iterate through the string and compare each char.
	// This is also the final stage of the comparison, so make sure R0 is
	// ready to return.

	MOV R0, #0				// Clear return register
	MOV R3, #0				// Clear an iterator
	MOV R1, R6				// Restore first address
	MOV R2, R7				// Restore second address
```

```
	1:
	LDRB R6, [R1, R3]		// Load into a temp register the char at index R3
	LDRB R7, [R2, R3]		// Load into a temp register the char at index R3
	CMP R6, R7			// Compare the two chars
	ITT NE				// If the chars are not equal
		MOVNE R0, #1		// Load 1 into the return register, aka not equal
		BNE return		// Return

	ADD R3, #1			// Increment iterator
	CMP R3, R5			// Compare incremented iterator to string length
	BGT return			// If the iterator is greater than the string length, return
					// That means all of the chars were equal.

	B 1b				// Otherwise keep looping

	return:
	POP {R1-R3, R5-R7, PC}


//	Function: ASCII_StringLength
//	Register-safe!
//	Description:
//		Determines the length of a null-terminated string in memory
//	Args:
//		R1	-	String address
//	Returns:
//		R0	-	Length
//	Register Use:
//
ASCII_StringLength:
	PUSH {R1, R2, LR}

	MOV R0, #0		// Clear iterator

1:
	LDRB R2, [R1, R0]	// Load character at index R0
	CMP R2, #0		// Determine if the char is null
	ITT NE
		ADDNE R0, #1	// Increment iterator if not zero
		BNE 1b

	POP {R1, R2, PC}
```