University of Texas at El Paso
Electrical and Computer Engineering Department

EE 4178/5190 – Laboratory for Microprocessors II

LAB 7

Peripherals and Queues: ADC and PWM (LEDC)

Objective:

- The objective for this lab is to understand how use the LEDC and ADC API's of espressif. In this lab, create 2 tasks: one that will initialize the peripherals and perform ADC readings every 100 millisecond. The ADC input reading should come from a 10K potentiometer and store its information into a queue. For the second task, output a PWM signal which gets its duty cycle updated based on the queue value send from the ADC task.

Bonus:

- For EE4178 is just a bonus and for EE5190 is mandatory
  - Add a port interrupt to stop and start the PWM signal.
- Bonus for EE5190
  - Create another task with two more PWM pins using the LEDC driver and start on at the highest duty cycle and then drop down. When it reaches 0 start the next pin from 0 to the highest duty cycle and go back doing the same pattern. This will give the illusion of a wave.

Pre-Lab:

- If you have a 12-bit resolution, what is the equation needed to convert from the raw ADC reading to Volts?
- What GPIO pins can you use for ADC1?
- What is the function to send out a queue?
- Using the LEDC API, what structures needs to be filled out to setup a PWM pin?
- What GPIO pins can be used for PWM?

By: Erick Baca and Jesus Minjares

C helpful functions

For this Lab, there are couple additional functions from ESPRESSIF that are important for using ADC. First is understanding what bit width are you planning to read using the function **adc1_config_width(adc_bits_width_t width_bit)**. The higher the bit width, the accurate the ADC reading will be.

- **typedef enum** {
  #if CONFIG_IDF_TARGET_ESP32
  ADC_WIDTH_BIT_9 = 0, /*!< ADC capture width is 9Bit. */
  ADC_WIDTH_BIT_10 = 1, /*!< ADC capture width is 10Bit. */
  ADC_WIDTH_BIT_11 = 2, /*!< ADC capture width is 11Bit. */
  ADC_WIDTH_BIT_12 = 3, /*!< ADC capture width is 12Bit. */
  #elif SOC_ADC_MAX_BITWIDTH == 12
  ADC_WIDTH_BIT_12 = 3, /*!< ADC capture width is 12Bit. */
  #elif SOC_ADC_MAX_BITWIDTH == 13
  ADC_WIDTH_BIT_13 = 4, /*!< ADC capture width is 13Bit. */
  #endif
  ADC_WIDTH_MAX,
  } **adc_bits_width_t;**

Next, you need to take into consideration that the ESP32 has specific pin for each channel for ADC1. **adc1_config_channel_atten(adc1_channel_t channel, adc_atten_t atten)** this functions is being use to declare which channel will you be reading from. Below you will see the channel numbers with its designated GPIO pins.

- **typedef enum** {
  ADC1_CHANNEL_0 = 0, /*!< ADC1 channel 0 is GPIO36 */
  ADC1_CHANNEL_1, /*!< ADC1 channel 1 is GPIO37 */
  ADC1_CHANNEL_2, /*!< ADC1 channel 2 is GPIO38 */
  ADC1_CHANNEL_3, /*!< ADC1 channel 3 is GPIO39 */
  ADC1_CHANNEL_4, /*!< ADC1 channel 4 is GPIO32 */
  ADC1_CHANNEL_5, /*!< ADC1 channel 5 is GPIO33 */
  ADC1_CHANNEL_6, /*!< ADC1 channel 6 is GPIO34 */
  ADC1_CHANNEL_7, /*!< ADC1 channel 7 is GPIO35 */
  ADC1_CHANNEL_MAX,
  } **adc1_channel_t;**

Lastly, in order to aquire the ADC reading you will use the function int **adc1_get_raw(adc1_channel_t channel).**

Now for the the LEDC API to create a PWM signal, to set up this function you will need to fill two structures. First you need to setup the structure **ledc_timer_config_t** where you put the duty resolution, timer, frequency, and clock source.

- **typedef struct** {
  ledc_mode_t speed_mode; /*!< LEDC speed speed_mode, high-speed mode or low-speed mode */
  union {
  ledc_timer_bit_t duty_resolution; /*!< LEDC channel duty resolution */
  ledc_timer_bit_t bit_num __attribute__((deprecated)); /*!< Deprecated in ESP-IDF 3.0. This is an alias to 'duty_resolution' for backward compatibility with ESP-IDF 2.1 */
  };
  ledc_timer_t timer_num; /*!< The timer source of channel (0 - 3) */
  uint32_t freq_hz; /*!< LEDC timer frequency (Hz) */
  ledc_clk_cfg_t clk_cfg; /*!< Configure LEDC source clock.For low speed channels and high speed channels, you can specify the source clock using LEDC_USE_REF_TICK, LEDC_USE_APB_CLK or LEDC_AUTO_CLK. For low speed channels, you can also specify the source clock using LEDC_USE_RTC8M_CLK, in this case, all low speed channel's source clock must be RTC8M_CLK*/

} **ledc_timer_config_t;**

Next, you will need to fill out the channel structure which is **ledc_channel_config_t** where you will select the pin number, speed mode, channel (which is the one we setup before), intr type, timer select (which use for the previuos timer structure), duty cycle and hpoint.

- **typedef struct** {
  int gpio_num; /*!< the LEDC output gpio_num, if you want to use gpio16, gpio_num = 16 */
  ledc_mode_t speed_mode; /*!< LEDC speed speed_mode, high-speed mode or low-speed mode */
  ledc_channel_t channel; /*!< LEDC channel (0 - 7) */
  ledc_intr_type_t intr_type; /*!< configure interrupt, Fade interrupt enable or Fade interrupt disable */
  ledc_timer_t timer_sel; /*!< Select the timer source of channel (0 - 3) */
  uint32_t duty; /*!< LEDC channel duty, the range of duty setting is [0, (2**duty_resolution)] */
  int hpoint; /*!< LEDC channel hpoint value, the max value is 0xfffff */
  struct {
  unsigned int output_invert: 1;/*!< Enable (1) or disable (0) gpio output invert */
  } flags; /*!< LEDC flags */
  } **ledc_channel_config_t;**

Then the functions to change the duty cycle is **ledc_set_duty(ledc_mode_t speed_mode, ledc_channel_t channel, uint32_t duty)** and to actually update the change you need the function **ledc_update_duty(ledc_mode_t speed_mode, ledc_channel_t channel).**

Finally the last important function while using the LEDC API is a way to pause the timer **ledc_timer_pause(ledc_mode_t speed_mode, ledc_timer_t timer_sel)** and to resume the timer **ledc_timer_resume(ledc_mode_t speed_mode, ledc_timer_t timer_sel).**