LED Controller Documentation

To control the brightness of LED we use PWM (Pulse Width Modulation) method. Pulse Width Modulation, or PWM, is a technique for getting analog results with digital means. Digital control is used to create a square wave, a signal switched between on and off. This on-off pattern can simulate voltages in between full on (3.3 Volts) and off (0 Volts) by changing the portion of the time the signal spends on versus the time that the signal spends off. The duration of “on time” is called the pulse width. To get varying analog values, you change, or modulate, that pulse width. If you repeat this on-off pattern fast enough with an LED for example, the result is as if the signal is a steady voltage between 0 and 3.3V controlling the brightness of the LED. In esp 32 we can configure the LED PWM controller to control the brightness of the LED.

To configure the LED PWM Controller, we need to do the following.

1. Configure settings in the structure led_timer_config_t. Pass a reference of this structure to ledc_timer_config().
2. Configure settings in the structure led_channel_config_t. Pass a reference of this structure to ledc_channel_config().

The LED PWM controller uses PWM to control the brightness of an LED. The frequency of the square wave used needs to be defined and also the resolution of

its duty cycle. Here, we have used a frequency 5Khz and duty resolution of 13 bits, which means that the duty cycle can be assigned wit a value in the range

0 to 2^n-1, where n= duty resolution. Here, the resolution used is 13 bits, so the range from 0 to 8191.

Higher the duty of PWM wave, more the brightness of the LED.

There are 16 channels in total. 8 for high speed timer and 8 for low speed timers.

Configure timer

Prepare and Set up timer for LED controller that will be used by LED Controller
Ledc_timer_config is the function where we configure the clock ,duty cycle,frequency and speed modesto control the led.

```
ledc_timer_config_t ledc_timer = {
    .duty_resolution = LEDC_TIMER_13_BIT,   // resolution of PWM duty
    .freq_hz = 5000,                        // frequency of PWM signal
    .speed_mode = LEDC_LS_MODE,             // timer mode
    .timer_num = LEDC_LS_TIMER,             // timer index
    .clk_cfg = LEDC_AUTO_CLK,               // Auto select the source clock
```

```
};
// Set configuration of timer0 for high speed channels
ledc_timer_config(&ledc_timer);
```

There are two modes in timer -> low speed mode and high speed mode. Note that the parameter clk_cfg has been set to LEDC_AUTO_CLK. This is done so that ESP32 automatically chooses one amongst the two internal clocks inside it according to the frequency, timer speed mode and the duy resolution.

Configure Channel

When the timer is set up, configure a selected channel (one out of `ledc_channel_t`). This is done by calling the function `ledc_channel_config()`.

Similar to the timer configuration, the channel setup function should be passed a structure `ledc_channel_config_t` that contains the channel's configuration parameters.

For example

```
ledc_channel_config_t ledc_channel = {

#ifdef CONFIG_IDF_TARGET_ESP32

    {

        .channel    = LEDC_CHANNEL_0,

        .duty       = 0,

        .gpio_num   = 2,

        .speed_mode = LEDC_HS_MODE,

        .hpoint     = 0,

       .timer_sel  = LEDC_HS_TIMER
```

```
    },
```

In this code ledc_channel is the name assigned for this channel, and within the structure definition we can see that there are various parameters to be declared, which includes duty cycle, gpio number, mode of operation, timerselect.

Parameters.

- `speed_mode`: Select the LEDC channel group with specified speed mode. Note that not all targets support high speed mode.
- `timer_num`: LEDC timer index (0-3), select from ledc_timer_t
- `timer_sel`: Timer index (0-3), there are 4 timers in LEDC module
- `clock_divider`: Timer clock divide value, the timer clock is divided from the selected clock source.
- `duty_resolution`: Resolution of duty setting in number of bits. The range of duty values is [0, (2**duty_resolution)]
- `clk_src`: Select LEDC source clock.
- `gpio_num`: The LEDC output gpio
- `ledc_channel`: LEDC channel (0-7), select from ledc_channel_t
- `duty`: Set the LEDC duty, the range of duty setting is [0, (2**duty_resolution)]
- `hpoint`: Set the LEDC hpoint value(max: 0xfffff)

The above mentioned parameters are used in this led timer and channel configuration.

LEDC High and Low Speed Mode

The advantage of high speed mode is glitch-free changeover of the timer settings. This means that if the timer settings are modified, the changes will be applied automatically on the next overflow interrupt of the timer. In contrast, when updating the low-speed timer, the change of settings should be explicitly triggered by software.
To configure these two modes we need to define whichever times and channels we are going to use to operate in high speed mode and low speed mode specifically.

```
if(ledc_mode_cfg->ledc_speed == LEDC_MODE_T__LEDC_MODE_E__LEDC_HIGH_SPEED_MODE)//Now,
{
    ledc_timer.speed_mode = LEDC_HIGH_SPEED_MODE;// set timer mode to high speed
    ledc_channel.speed_mode = LEDC_HIGH_SPEED_MODE;// set channel mode to high speed
    ledc_timer.timer_num = LEDC_TIMER_0;// use TIMER 0 for high speed purposes
    ledc_channel.timer_sel = LEDC_TIMER_0;// The channel will use TIMER 0
```

ledc_timer.speed_mode = LEDC_HIGH_SPEED_MODE;// set timer mode to high speed

ledc_channel.speed_mode = LEDC_HIGH_SPEED_MODE; // set channel mode to high speed

ledc_timer.timer_num = LEDC_TIMER_0;   // This means that we are going to use TIMER 0 for High speed mode

And then we will assign this timer to the channel we going to use for high speed mode
ledc_channel.timer_sel = LEDC_TIMER_0;

And if you want to configure it for low speed mode put LEDC_LOW_SPEED_MODE instead of LEDC_HIGH_SPEED_MODE. But we need to assign another timer for LEDC_LOW_SPEED_MODE , say TIMER1(ledc_timer.timer_num = LEDC_TIMER_1) and then set this timer to the channel we going to use for low speed mode.

LEDC_SPEED_MODE_MAX can be used in place of LEDC_LOW_SPEED_MODE to get the maximum speed mode.

After configuring the above mentioned parameters we have to pass these structures to function to set the timer.
ledc_timer_config(&ledc_timer).

By using switch..case we assign the channel by providing user input as shown below..
The user input a number from 0 to 7 in the client. Here we assign the appropriate enum value corresponding to the desired channel.

```
switch(ledc_channel_cfg->channel)//Now, we set the channel according to user request
{
    case 0:
        ledc_channel.channel = LEDC_CHANNEL_0;
        break;
    case 1:
        ledc_channel.channel = LEDC_CHANNEL_1;
        break;
    case 2:
        ledc_channel.channel = LEDC_CHANNEL_2;
        break;
```

Once the channel starts operating and generating the PWM signal with the constant duty cycle and frequency, there are a couple of ways to change this signal.The range of

the duty cycle values passed to functions depends on selected `duty_resolution` and should be from `0` to `(2 ** duty_resolution) - 1`.

In the ledcontroller_func(void *clnt, void *request) in app_main.c; the parameters assigned to the request field in received protobuf message is passed to it as a parameter named request. Since it is a pointer of type (void *), we cannot access any values inside it. Therefore it has to be converted to pointer of type (Request *) corresponding to Request field structure(check messagefile.pb-c.h) in order to retrieve data from it.

```
ledc_channel.duty = ((Request*)request)->ledc_channel_config_request->ledc_conf->duty;// set the d
ledc_channel.gpio_num = ((Request*)request)->ledc_channel_config_request->ledc_conf->gpio_num;// s
ledc_channel.hpoint = ((Request*)request)->ledc_channel_config_request->ledc_conf->hpoint;// Read
```

.

Additions made to messagefile.proto

We shall add a third response and request message type, which we will call ledc_channel_config_Request and led_channel_config_Response. ledc_channel_config_Request will have a submessage ledc_conf of type ledc_channel_config_t, which contains the parameters such as timer speed mode, channel number, gpio pin number, etc. The message ledc_channel_config_Response will have a parameter of integer type called status, which will be assigned the value 1 in case of successful operation in the ESP32.

Inside the oneof parameter RequestFunc in message Request, we add a submessage as shown below:

```
ledc_channel_config_Request ledc_channel_config_request = 3;
```

Inside the oneof parameter ResponseFunc in message Response, we add another submessage

```
ledc_channel_config_Request ledc_channel_config_request = 3;
```

Then, we define the parameters inside each of these messages.

```
message ledc_channel_config_Request{
        ledc_channel_config_t ledc_conf = 1;
}
message ledc_channel_config_Response{
        int32 config_status = 1;
}
```

The message type ledc_channel_config_type_t contains parameters to configure the gpio pin number(gpio_num), timer speed mode(speed_mode), channel number (channel), interrupt type(intr_type) , duty cycle(duty) and hpoint, which are required to configure the timer and the ledc channel.

```
message ledc_channel_config_t{
    int32 gpio_num = 1;
    ledc_mode_t speed_mode = 2;
    ledc_channel_t channel = 3;
    ledc_intr_type_t intr_type = 4;
    uint32 duty = 5;
    int32 hpoint = 6;
}
```

The parameter speed_mode is stored as a submessage named ledc_mode_t; in which we store the timer speed mode as an enumerator type which has the values shown below. The client will set the enumerator value to ledc_mode_e according to the user input.

```
message ledc_channel_t{
        int32 channel = 1;
}
message ledc_mode_t{
    enum ledc_mode_e {
        LEDC_HIGH_SPEED_MODE = 0;
        LEDC_LOW_SPEED_MODE = 1;
        LEDC_SPEED_MODE_MAX = 2;
        }
        ledc_mode_e ledc_speed = 1;
}
```

The parameter channel is stored in a submessage named ledc_channel_t, which contains a parameter channel of integer type, which is used to store the channel no from 0 to 8 as provided by the user. Later, it shall be converted to an appropriate enumerator value used by the ESP32 in our ESP32 software(app_main.c)
The enumerator types thus created have to be declared as submessages so that they can be accessed by the program.

Similarly, we also add another message named ledc_intr_type.

```
message ledc_intr_type_t{
        enum interrupt_type{
                LEDC_INTR_DISABLE = 0;
                LEDC_INTR_FADE_END = 1;
        }
        interrupt_type intr = 1;
}
```

There are 2 interrupt types used by the ESP32 ledcontroller; namely LEDC_INTR_DISABLE and LED_INTR_FADE_END , corresponding to disable Fade interrupt or enable Fade interrupt.