PWM Driven Servomotor Combined With LED Blinking Using Kernel Threading & FIFO

This example deals with controlling an on-board LED and a PWM driven servomotor simultaneously using the concept of multitasking. Refer to Kernel Services for a deeper understanding.

This example demonstrates spawning multiple threads using **K_THREAD_DEFINE()**. It spawns three threads. Each thread is then defined at compile time using **K_THREAD_DEFINE()**.

One of the threads controls an LED while the other controls the servometer. The third thread uses **printk()** to print the information added to the FIFO to the device console (optional).

When either of these threads toggles its LED and servomotor respectively, it also pushes information into a FIFO to identify the count of number of times toggled.

Let us look at the application source code to understand the precise working of this example.

```
#include <zephyr.h>
#include <device.h>
#include <drivers/gpio.h>
#include <sys/printk.h>
#include <sys/__assert.h>
#include <string.h>
#include <drivers/pwm.h>
```

First, we include all the required header files for this program. Note that inclusion of PWM driver is essential.

```
/* size of stack area used by each thread */
#define STACKSIZE 1024

/* scheduling priority used by each thread */
#define PRIORITY 7

/*Newly added*/
#define PERIOD_USEC (20U * USEC_PER_MSEC)
#define STEP_USEC 100
#define MIN_PULSE_USEC 700
#define MAX_PULSE_USEC 2300
```

We also define the stack size (Bytes) which can be used by the threads and a scheduling priority for the threads (can be changed at compile time).

Next we define the macros for the time durations required for the PWM driven servomotor.

```
#define LED1_NODE DT_ALIAS(led1)
#define PWM_NODE DT_ALIAS(pwm_servo)
#if !DT_NODE_HAS_STATUS(LED1_NODE, okay)
#error "Unsupported board: led1 devicetree alias is not defined"
#endif

#if !DT_NODE_HAS_STATUS(PWM_NODE, okay)
#error "Unsupported board: pwm devicetree alias is not defined"
#endif
```

DT_ALIAS() gets a node identifier from the aliases which are respectively defined as LED1_NODE and PWM_NODE.

Extremely Important: Do not use the *led*0 node for the *nrf*52840*dk_nrf*52811 board for this application. For some reason, the *led*0 is used by the servomotor and it is ON throughout the duration of servomotor being on. For that reason, the *led*1 is used in this example.

Next, we need to check if the obtained node identifier (for both LED and the servomotor) has it's status enabled or disabled. Accordingly, an error message has to be displayed on to the console.


```
enum direction {
    DOWN,
    UP,
};

struct pwm_data_t {
    void *fifo_reserved;
    uint32_t count;
};

struct printk_data_t {
    void *fifo_reserved; /* 1st word reserved for use by fifo */
    uint32_t led;
    uint32_t cnt;
};
```

To keep track of the direction of the servomotor, we use an enumerator for it.

Next we define two structures *pwm_data_t* and *printk_data_t* representing the type of information that goes into the FIFO. While the *printk_data_t* has information regarding the LED being ON or OFF and the number of times the LED has been toggled, the *pwm_data_t* doesn't hold much useful information. It just has a count variable which is of practically no significance. It has been declared just to push some data from the PWM node to the FIFO.

```
K_FIFO_DEFINE(printk_fifo);

struct led {
struct gpio_dt_spec spec;
const char *gpio_pin_name;
};

static const struct led led1 = {
    .spec = GPIO_DT_SPEC_GET_OR(LED1_NODE, gpios, 0),
    .gpio_pin_name = DT_PROP_OR(LED1_NODE, label, ""),
};
```

K_FIFO_DEFINE() statically defines and initializes a FIFO queue with the name *printk_fifo*.

We also need to create a structure for the *led*. **gpio_dt_spec** provides a type to hold GPIO information specified in devicetree. We also declare a pointer pointing to the GPIO pin name.

Next we create a variable *led1* of the datatype *led* as defined earlier. **GPIO_DT_SPEC_GET_OR()** returns a static initializer for a gpio_dt_spec structure given a devicetree node identifier, a property specifying a GPIO and an index. **DT_PROP_OR()** gets a devicetree property value.


```
void servo(uint32_t sleep_ms)
{
    const struct device *pwm;
    uint32_t pulse_width = MIN_PULSE_USEC;
    enum direction dir = UP;
        int count = 0;
    int ret;

    pwm = DEVICE_DT_GET(PWM_NODE);

    if (!device_is_ready(pwm)) {
        printk("Error: PWM device %s is not ready\n", pwm->name);
        return;
    }
```

Now, we define a function (which returns nothing) to control the servomotor. It takes the time in milliseconds as it's parameter. The basics of working of PWM servomotor has been explained in the PWM driven servomotor section.

```
while (1) {

        struct pwm_data_t data = { .count = count };

        size_t size = sizeof(data);
        char *ptr = k_malloc(size);
        __ASSERT_NO_MSG(ptr != 0);
        memcpy(ptr, &data, size);
        k_fifo_put(&printk_fifo, ptr);
        count++;
```

In the code snippet given above, an infinite loop is initialized and a variable of the datatype *pwm_data_t* is declared with it's property count equated to the variable *count*.

The next few lines correspond to pushing this *count* information to the FIFO defined earlier by utilizing the *Memory Management Functions*.


```
ret = pwm_pin_set_usec(pwm, DT_PROP(DT_NODELABEL(pwm0), ch0_pin), PERIOD_USEC, pulse_width, 0);

if (ret < 0) {
    printk("Error %d: failed to set pulse width\n", ret);
    return;
}

if (dir == DOWN) {
    if (pulse_width <= MIN_PULSE_USEC) {
        dir = UP;
        pulse_width = MIN_PULSE_USEC;
    } else {
        pulse_width -= STEP_USEC;
    }
} else {
        pulse_width += STEP_USEC;
    if (pulse_width >= MAX_PULSE_USEC) {
            dir = DOWN;
            pulse_width = MAX_PULSE_USEC;
        }
    }
            k_msleep(sleep_ms);
        }
}
```

The above snippet corresponds to the control of the PWM driven

servomotor which has been explained in the section PWM driven servomotor before.

```
void blink(const struct led *led, uint32_t sleep_ms, uint32_t id)
{
    const struct gpio_dt_spec *spec = &led->spec;
    int cnt = 0;
    int ret;

    if (!device_is_ready(spec->port)) {
        printk("Error: %s device is not ready\n", spec->port->name);
        return;
    }

    ret = gpio_pin_configure_dt(spec, GPIO_OUTPUT);

    if (ret != 0) {
        printk("Error %d: failed to configure pin %d (LED '%s')\n", ret, spec->pin,
        led->gpio_pin_name);
        return;
    }
```

Here, we define a function to separately control the blinking of the LED. We declare a variable *cnt* for counting and a return value to verify and validate the configuration of the GPIO pin.

```
while (1) {
        gpio_pin_set(spec->port, spec->pin, cnt % 2);

        struct printk_data_t tx_data = { .led = id, .cnt = cnt };

        size_t size = sizeof(struct printk_data_t);
        char *mem_ptr = k_malloc(size);
        __ASSERT_NO_MSG(mem_ptr != 0);

        memcpy(mem_ptr, &tx_data, size);

        k_fifo_put(&printk_fifo, mem_ptr);

        k_msleep(sleep_ms);
        cnt++;
    }
}
```

In the infinite loop, we use the function **gpio_pin_set()** to set logical level of an output pin. We take the remainder value of the division of *cnt* and 2 as it is always either 0 or 1. We push this count and LED status information into the FIFO, give a delay and increment the *cnt* for the loop to repeat.

```
void blink1(void)
{
    blink(&led1, 2000, 0);
}

void servo0(void)
{
    servo(50);
}
```

This is where we call the above defined functions by passing the desired values of time delays. The LED will blink at a rate of 2 seconds while the servomotor rotates at the rate of 50 milliseconds.


```
void uart_out(void)
{
    while (1) {
        struct printk_data_t *rx_data0 = k_fifo_get(&printk_fifo,
                                K_FOREVER);
        struct pwm_data_t *rx_data1 = k_fifo_get(&printk_fifo,
                                K_FOREVER);
        printk("Toggled led%d; counter=%d\n; pwm counter=%d\n",
               rx_data0->led, rx_data0->cnt, rx_data1->count);
        k_free(rx_data0);
        k_free(rx_data1);
    }
}
```

This is the function written to display the contents of the FIFO as they happen. It is optional. We declare a variable each of datatype *printk_data_t* and *pwm_data_t* which are *rx_data*0 and *rx_data*1. The content of these variables are fetched from the FIFO using the kernel function **k_fifo_get()**. Once we print these out on to the console, we need to free these variables.

```
K_THREAD_DEFINE(blink1_id, STACKSIZE, blink1, NULL, NULL, NULL, PRIORITY, 0, 0);
K_THREAD_DEFINE(uart_out_id, STACKSIZE, uart_out, NULL, NULL, NULL, PRIORITY, 0, 0);
K_THREAD_DEFINE(servo0_id, STACKSIZE, servo0, NULL, NULL, NULL, PRIORITY, 0, 0);
```

As explained earlier, **K_THREAD_DEFINE()** statically defines and initializes a thread. The parameters it takes are

- name – Name of the thread.
- stack_size – Stack size in bytes.
- entry – Thread entry function.
- $p1$ – 1st entry point parameter.
- $p2$ – 2nd entry point parameter.
- $p3$ – 3rd entry point parameter.
- prio – Thread priority.
- options – Thread options.
- delay – Scheduling delay (in milliseconds), zero for no delay.

To observe the output using hardware, we need

- $nRF52840$ DK
- $SG90$ Micro Servo
- Jumper wires

Power-up the board with a micro USB cable and make the connections as shown in the table below:

$SG90$	$nRF52840DK$
Vcc	$5V$
GND	GND
SIG	$P0.13$

Table 3: Pin connections for $SG90$ with $nRF52840DK$

To view the output, run the commands:

```
west build -p auto -b nrf52840dk_nrf52811 mt_pwm_servo_led
west flash --erase
```