CSE-522 REAL-TIME EMBEDDED SYSTEMS

Assignment-2 Report

Raunak-1217240245
March 5, 2021

1 HCSR04 DRIVER IMPLEMENTATION

1.1 DEVICE_AND_API_INIT

```c
#define DEVICE_AND_API_INIT(dev_name, drv_name, init_fn, data, cfg_info,
          level, prio, api)                                           \
  static struct device_config _CONCAT(__config_, dev_name) __used    \
  __attribute__((__section__(".devconfig.init"))) = {                 \
    .name = drv_name, .init = (init_fn),                              \
    .config_info = (cfg_info)                                         \
  };                                                                  \
  static struct device _CONCAT(__device_, dev_name) __used            \
  __attribute__((__section__(".init_" #level STRINGIFY(prio)))) = {  \
    .config = &_CONCAT(__config_, dev_name),                          \
    .driver_api = api,                                                \
    .driver_data = data                                               \
  }
```

It creates a device object and configures it for initialization during the boot time[1] as well as it sets the driver_api[2]. It invokes the device_init, which defines a device object, which is configured by the kernel during system initialization. In our implementation, we use this structure to specify:

- initialization function, init_fn
- pointer to the our device structure, data
- initilization level and priority of our device
- our driver api

These all are initialized at the start and this macro places the object in memory through the linker at the point of system initialization.

In our driver we have used this macro as follows,

```
DEVICE_AND_API_INIT(HC_SR04_00, "HCSR0", hc_sr04_init, &hcsr0_data,
        NULL, POST_KERNEL, CONFIG_SENSOR_INIT_PRIORITY,
        &hc_sr04_driver_api);
```

where HC_SR04_00 defines name of the device and HCSR0 defines name of the driver. hcsr0_data is our per device structure, cfg_info is null in our case and level is set to POST_KERNEL, which means that this device requires kernel services during configuration and CONFIG SENSOR INIT PRIORITY defines our priority integer value, lower values indicates earlier initialization.

Whenever we use this macro it calls hc_sr04_init, sets level, priority and maps the api at the start. Here, the fourth parameter hcsr0_data is our user defined object of type user defined data structure per device of type, struct hc_sr04_data.

Now, we will define the initialization function and how our hcsr04 device is initialized in the driver.

1.2 HCSR04 DEVICE INITIALIZATION

In our hcsr04 init function we pass our device structure as argument.

```
static int hc_sr04_init(struct device *dev)
```

```
struct device {
  struct device_config *config;
  const void *driver_api;
  void *driver_data;
  };
```

From this device structure[1] we obtain the driver_data structure and store in a pointer variable drv_data of type struct hc_sr04_data.

```
struct hc_sr04_data *drv_data = dev->driver_data;
```

Our user defined per device hcsr04 data structure consists of the following elements:

```
struct hc_sr04_data {
  struct device *gpio;
  struct gpio_callback gpio_cb;
  u8_t trigger;
  u8_t echo;
  u8_t status;
  unsigned long long start;
  unsigned long long end;
  u16_t distance;
  u32_t timestamp;
  u16_t timeout;
  struct k_sem sem;
};
```

Then we use device_get_binding to get the device structure. It searches for the device using the device's configuration name. It returns a pointer to the device structure, if found else, it returns NULL. If we want to retrieve the device from the memory, where it is placed by the linker at initialization, we can use device get binding to get the required pointer to the device structure.

```
pinmux = device_get_binding(CONFIG_PINMUX_NAME);
```

```
struct galileo_data *dev1 = pinmux->driver_data;
drv_data->gpio = dev1->gpio_dw;
```

We extract the galileo_data structure from the pinmux device and then use the galileo data struct to get the gpio_dw, which represents GPIO0 to GPIO7(qrk_gip_gpio). galileo_data structure contains galileo specific data, it contains device structures corresponding to galileo devices, GPIOs, drivers and expanders.

```
struct device_config {
  const char *name;
  int (*init)(struct device *device);
  };
```

As we can see above[1] that in the device structure we have device_config and we can use our device structure to extract device_config, from which we can extract the name of our particular driver. we can use that name to find out which of the two hcsr04 devices is being initialized and we can set its trigger and echo pin configuration accordingly. For example,

```
if (strcmp((dev->config->name), "HCSR0")){

        drv_data->trigger = CONFIG_HCSR0_TRIGGER_PIN;
        drv_data->echo = CONFIG_HCSR0_ECHO_PIN;

    }
```

Now, we utilize pinmux_pin_set api to set our echo and trigger pin direction to input and ourtput respectively. pinmux_pin_set is used to set the gpio pins to in, out or as pwm and other galileo gpio pins. It is defined as follows[1],

```
static inline int pinmux_pin_set(struct device *dev, u32_t pin, u32_t
    func)
{
  const struct pinmux_driver_api *api =
    (const struct pinmux_driver_api *)dev->driver_api;

  return api->set(dev, pin, func);
}
```

Further, we will initialize the echo pin as the interrupt pin using gpio_pin_configure. gpio pin configure is used to set the direction of the pin based on a flag parameter. It returns gpio_config value indicating whether the pin is configures or not, a negative value represents an error in the configuration. It is defined as follows[1],

```
static inline int gpio_pin_configure(struct device *port,u32_t pin,int
    flags)
{
```

```c
  return gpio_config(port, GPIO_ACCESS_BY_PIN, pin, flags);
}
```

Now, we will setup the gpio callback using gpio init callback, gpio add callback and gpio pin enable callback. These are explained below[1],

```c
static inline void gpio_init_callback(struct gpio_callback *callback,
              gpio_callback_handler_t handler, u32_t pin_mask)
{
  __ASSERT(callback, "Callback pointer should not be NULL");
  __ASSERT(handler, "Callback handler pointer should not be NULL");
  callback->handler = handler;
  callback->pin_mask = pin_mask;
}
```

It is a helper function to initialize gpio_callback. It uses ASSERT macro to check for pointers validity.

```c
static inline int gpio_add_callback(struct device *port,struct
    gpio_callback *callback)
{
  const struct gpio_driver_api *api =
    (const struct gpio_driver_api *)port->driver_api;
  if (api->manage_callback == NULL) {
    return -ENOTSUP;
  }
  return api->manage_callback(port, callback, true);
}
```

This API adds an application callback[1]. It utilizes driver api to manage the callback and returns the output of manage callback function, which adds the application callback. A returned negative value indicates error in the configuration.

```c
static inline int gpio_pin_enable_callback(struct device *port,u32_t pin)
{
  return gpio_enable_callback(port, GPIO_ACCESS_BY_PIN, pin);
}
```

This API enables the trigger of the handler function when interrupted. A returned negative value indicates error in the configuration[1].

1.3 SENSOR_DRIVER INITIALIZATION

Now we define our sensor apis, sample fetch, channel get and attribute set functions and map it to sensor driver api, as depicted below,

```c
static const struct sensor_driver_api hc_sr04_driver_api = {
  .sample_fetch = hc_sr04_sample_fetch,
  .channel_get = hc_sr04_channel_get,
  .attr_set = hc_sr04_attr_set,
};
```

It consists of various typedefs, which represents different sensor apis, which we connect to our user defined sensor driver api using sensor_driver_api structure, which we map and pass it to the device initialization macros such as, DEVICE_INIT and DEVICE_AND_API_INIT.

hc_sr04_sample_fetch, hc_sr04_channel_get and hc_sr04_attr_set are our user defined functions. Different sensor driver apis are explained below,

1.3.1 SENSOR_SAMPLE_FETCH

```
int sensor_sample_fetch_chan(const struct device *dev, enum
    sensor_channel type)
```

This API fetches samples from the driver and store the value in an internal buffer[3]. It consists of following parameters:

- **dev**: Device structure of the sensor.
- **type**: Type of the channel.

1.3.2 SENSOR_CHANNEL_GET

```
int sensor_channel_get(const struct device *dev, enum sensor_channel chan
    , struct sensor_value *val)
```

This API returns an useful measurement of the sensor on a particular channel from the driver's internal buffer[3]. Prior to calling this function sensor_sample_fetch should be called to store the measurement in the driver's internal buffer. It consists of the following parameters:

- **dev**: Device structure of the sensor.
- **chan**: Type of the channel.
- **val**: Location to store the value.

1.3.3 SENSOR_ATTRIBUTE_SET

```
int sensor_attr_set(const struct device *dev, enum sensor_channel chan,
    enum sensor_attribute attr, const struct sensor_value *val)
```

This API sets the attribute of the sensor[3]. It consists of the following parameters:

- **dev**: Device structure of the sensor.
- **chan**: Type of the channel.
- **attr**: Type of attribute to retrieve.
- **val**: Location to store the value.

2 TEST-LED IMPLEMENTATION

2.1 Devices Created and order of initialization:

1. Device of the type CONFIG_PINMUX_NAME.

```
pinmux=device_get_binding(CONFIG_PINMUX_NAME);

```

2. Device of the type GPIO_DW driver struct.

```
struct galileo_data *dev = pinmux->driver_data;
gpiob=dev->gpio_dw;

```

3. Device of the type CONFIG_PWM_PCA9685_0_DEV_NAME.

```
pwm = device_get_binding(CONFIG_PWM_PCA9685_0_DEV_NAME);

```

2.1.1 CONFIG_PINMUX_NAME

This device structure is used for pin multiplexing. In our test_led program, we have used this function to define the RGB LED pins as the pwm pin and interrupt pin as the input pin. For example,

```
pinmux_pin_set(pinmux,5,PINMUX_FUNC_C);
```

Here, we have defined IO pin 5 as pwm.led 3 pin.

2.1.2 GPIO_DW

This device structure is used to configure the gpio pins in the range from GPIO0 to GPIO7(qrk gip gpio), which is used to configure the pin as the interrupt pin and to configure the callback. For example,

```
gpio_pin_configure(gpiob, 7, GPIO_DIR_IN | GPIO_INT | EDGE_RISING);
gpio_init_callback(&gpio_cb, interrupt_cb, BIT(7));
```

Here, we are configuring gpio7 as the interrupt pin which is interrupted on the rising edge. Using init callback, we are configuring our interrupt handler function to call on an interrupt.

2.1.3 CONFIG_PWM_PCA9685_0_DEV_NAME

This device structure is used to configure the pwm period and the length of pulses to control the brightness of the led. For example,

```
pwm_pin_set_cycles(pwm, 3, 4096, 4096);
```

Here, we are applying 100 percent pwm signal as the total period is 4096 cycles and pulse period is also 4096 cycles on pwm.led3 pin.

References

[1] “device.h - include/device.h - zephyr source code (v1.14.2).” [Online]. Available: https://elixir.bootlin.com/zephyr/v1.14.2/source/include/device.h

[2] “Device driver model,” Oct 2019. [Online]. Available: https://docs.zephyrproject.org/1.14.1/reference/drivers/index.html#driver-data-structures

[3] “Sensors,” Mar 2021. [Online]. Available: https://docs.zephyrproject.org/latest/reference/peripherals/sensor.html#group__sensor__interface_1ga8bc61bc2dfaa6a911c068c4f9870d637