CSE 522: RTES – Assignment 2 Report

- **Author:** Viraj Savaliya
- **ASU ID:** 1217678787
- **Date:** 03/05/2021

To start developing a device driver in zephyr, first it needs to be initialized and configured in the system. The device initialization starts using ***DEVICE_AND_API_INIT()*.** The initialization macro is defined as ***DEVICE_AND_API_INIT(dev_name, drv_name, init_fn, data, cfg_info, level, prio, api).***

In the macro, the arguments to be provided are:

1. ***dev_name:*** This is the device name to be given to look up for the device when required.
2. ***drv_name:*** This is the driver's name which should be configured in kconfig file and expose to entire system.
3. ***init_fn:*** This is the initialization function to be assigned for the driver and will initialize the driver when systems boots.
4. ***data:*** This is the driver's data structure containing all the data configurations.
5. ***cfg_info:*** This is the structure containing all configuration information for the specific instance of the driver.
6. ***level:*** The level at which configuration occurs. This is set based on the dependencies and kernel service requirements of the driver.
7. ***prio:*** The priority of initialization for the device.
8. ***api:*** The initial pointer to the API function struct for the device.

Alternatively, we can also use ***DEVICE_INIT(dev_name, drv_name, init_fn, data, cfg_info, level, prio)*** if we don't have any API functions to be associated with the device. Both macros are similar, ***DEVICE_INIT*** is basically calling ***DEVICE_AND_API_INIT*** with API function struct argument as NULL. The macro basically defines a runtime per device struct by the device name specified which contains pointer to a device configuration struct, API struct and data struct for the device.

Once the device is initialized, we have all the things which are required to start for a device in zephyr. The first function to run for the device will be the init function specified. For e.g. in HCSR04 sensor driver when we define a device (HCSR0 or HCSR1), ***hcsr_init*** will execute when system boots to initialize the device. For the device here first thing we need is the data struct of that particular device instance. This can be obtained from the device pointer parameter passed to the init function ***hcsr04_init(struct device *dev).*** The init function gets the pointer of the device struct defined by the device name. This pointer of device contains a pointer to its data struct as explained before. Therefore, by using ***dev->driver_data*** we can access the data struct for that device instance of the sensor. Also, the name of the device can be checked using ***dev->config->name*** as the driver name provided during the initialization is stored in the device_config struct which can be accessed from the struct device. Now the data attributes for the device instant can be initialized which should be present at boot time. Also, GPIO pin configurations for the sensor attached to the board are done in the init function.

```
/**
 * @brief Static device information (In ROM) Per driver instance
 *
 * @param name name of the device
 * @param init init function for the driver
 * @param config_info address of driver instance config information
 */
struct device_config {
        const char *name;
        int (*init)(struct device *device);
#ifdef CONFIG_DEVICE_POWER_MANAGEMENT
        int (*device_pm_control)(struct device *device, u32_t command,
                                 void *context, device_pm_cb cb, void *arg);
        struct device_pm *pm;
#endif
        const void *config_info;
};

/**
 * @brief Runtime device structure (In memory) Per driver instance
 * @param device_config Build time config information
 * @param driver_api pointer to structure containing the API functions for
 * the device type. This pointer is filled in by the driver at init time.
 * @param driver_data driver instance data. For driver use only
 */
struct device {
        struct device_config *config;
        const void *driver_api;
        void *driver_data;
#if defined(__x86_64) && __SIZEOF_POINTER__ == 4
        /* The x32 ABI hits an edge case.  This is a 12 byte struct,
         * but the x86_64 linker will pack them only in units of 8
         * bytes, leading to alignment problems when iterating over
         * the link-time array.
         */
        void *padding;
#endif
};
```

Fig 1. struct device and struct device_config present in /include/device.h [1]

To get a pointer to struct device for any device defined in the zephyr kernel we can use ***device_get_binding(const char *name)*** which returns a pointer to the device struct having a name associated with it same as the name argument passed to it. This function basically parses through the device list of all devices defined and returns the respective device structure if found and NULL if not. The pinmux driver present on board can be found using ***CONFIG_PINMUX_NAME*** as all the drivers in zephyr are configured with some name associated with it. We will obtain a pointer to the device structure for pinmux and then will also be able to access the driver data structure of pinmux as explained before.

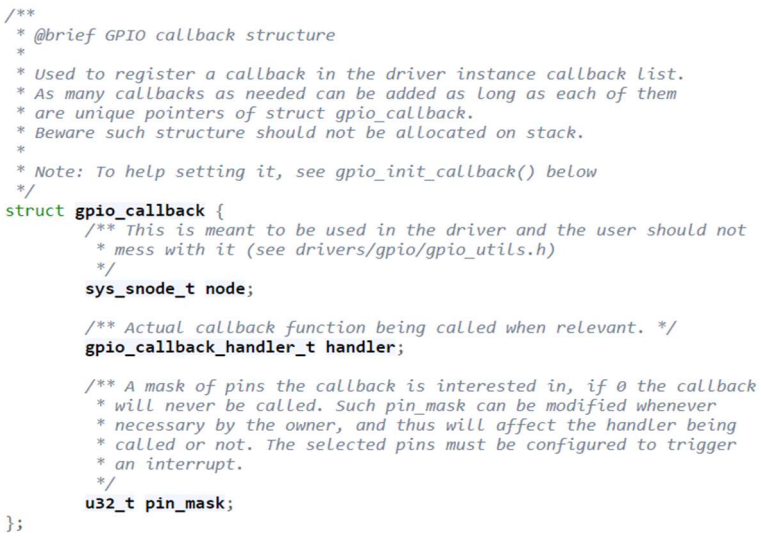
All the pins from different chips present on galileo are multiplexed in the limited IO pins present on board. To set a particular GPIO, PWM or any other internal pin of galileo you will need to access the device struct initialized by their name. All the device struct for different pins from different chips can be accessed via the ***struct galileo_data*** present in pinmux_galileo.h. The pinmux driver struct is basically the galileo_data structure and the required chip can be accessed through it. The Quark GIP Controller is present by the name ***GPIO_0*** and with the driver struct as ***gpio_dw*** which has 8 gpio pins. After getting the pinmux device structure using ***device_get_binding(CONFIG_PINMUX_NAME)***, we will be using ***pinmux->driver_data*** to get the galileo_data structure and then to access the GIP GPIO Controller on board we will use ***dev->gpio_dw*** , where dev is the galileo_data structure pointer obtained.

```
struct galileo_data {
        struct device *exp0;
        struct device *exp1;
        struct device *exp2;
        struct device *pwm0;

        /* GPIO<0>..GPIO<7> */
        struct device *gpio_dw;

        /* GPIO<8>..GPIO<9>, which means to pin 0 and 1 on core well. */
        struct device *gpio_core;

        /* GPIO_SUS<0>..GPIO_SUS<5> */
        struct device *gpio_resume;

        struct pin_config *mux_config;
};
```

Fig 2. struct galileo_data (pinmux driver_data) present in /boards/x86/galileo/pinmux_galileo.h [1]

The pinmux and gpio_dw device structure pointers can be used when accessing the APIs for pinmux and gpio and to perform operations on the desired pins being used by the sensor. For using any IO pins present on board we will have to set the IO pins to its desired internal pin of chips present on board. For the assignment 2, we are using only GPIO pins for the sensor. Two things we need to set and configure for any IO pins we use on board. First, we need to set the IO pin to a desired chip on board and then set the desired internal pin present on that board. To set the IO pin we need to refer to /boards/x86/galileo/pinmux.c to see what each IO pin refers to in the pin_config struct and then accordingly select the function for that IO pin. We use ***pinmux_pin_set(struct device *dev, u32_t pin, u32_t func)*** API to set an IO pin. The pointer to the pinmux for galileo is obtained by including the pinmux_galileo.h header and then using ***device_get_binding(CONFIG_PINMUX_NAME).*** Now the pinmux pointer is pointing to the pinmux device structure of galileo and in this way we identify the pinmux for galileo board. Now when using ***pinmux_pin_set()*** we provide the pinmux pointer, the IO pin number and the respective function we want to set for that pin. The ***pinmux_pin_set()*** will then access the ***pin_config*** struct via the ***galileo_data*** struct and then select the desired function for the IO pin.

Now to configure and use the internal pin e.g. gpio pin in this case we need to first configure the gpio pin using ***gpio_pin_configure(struct device *port, u32_t pin, int flags)*** where device is the ***gpio_dw*** device structure we obtained earlier and then pass the GPIO pin number and its flag value to set the GPIO for a particular mode (e.g. GPIO_DIR_OUT, EDGE_RISING, etc.). Now this will transfer the request to the ***gpio_dw_config(struct device *port, int access_op, u32_t pin, int flags)*** which is present in gpio_dw.c to configure the gpio pin present in the Quark GIP GPIO controller chip and then it calls the inline asm_inline_gcc functions to complete the request. After the pins are configured they can be written to or read from using the ***gpio_pin_write()*** and ***gpio_pin_read()*** which also work in a similar way as ***gpio_pin_configure()*** and all three generate a system call to move ahead.

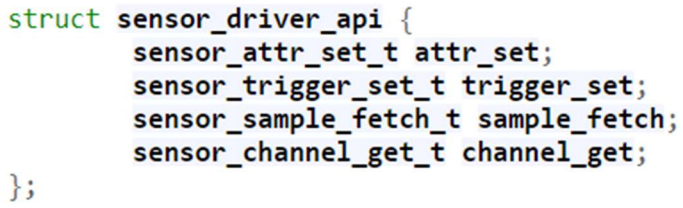


```
/**
 * @brief GPIO callback structure
 *
 * Used to register a callback in the driver instance callback list.
 * As many callbacks as needed can be added as long as each of them
 * are unique pointers of struct gpio_callback.
 * Beware such structure should not be allocated on stack.
 *
 * Note: To help setting it, see gpio_init_callback() below
 */
struct gpio_callback {
        /** This is meant to be used in the driver and the user should not
         * mess with it (see drivers/gpio/gpio_utils.h)
         */
        sys_snode_t node;

        /** Actual callback function being called when relevant. */
        gpio_callback_handler_t handler;

        /** A mask of pins the callback is interested in, if 0 the callback
         * will never be called. Such pin_mask can be modified whenever
         * necessary by the owner, and thus will affect the handler being
         * called or not. The selected pins must be configured to trigger
         * an interrupt.
         */
        u32_t pin_mask;
};
```

Fig 3. struct gpio_callback present in include/gpio.h [1]

Now, when using interrupt-based approach for monitoring a pin and then triggering an interrupt to perform some callback activity we will have to initialize and add a callback to check a particular gpio pin. To add a callback, we need to use the ***gpio_callback*** struct present in gpio.h file. We can initialize the callback using ***gpio_init_callback(struct gpio_callback *callback, gpio_callback_handler_t handler, u32_t pin_mask)*** where we provide the gpio_callback structure we define in the sensor, the callback function or the interrupt service routine handler function when callback is initiated and the bit mask for the gpio pin number we want to set a callback for (In this assignment, echo pin in HCSR04 sensor). The init function just populates the callback structure. Then we need to add the callback structure to the gpio_dw structure for the device created. This is done using ***gpio_add_callback(struct device *port, struct gpio_callback *callback)*** where the arguments passed are the device's gpio_dw struct and the address of the callback struct to be added. This callback can be enabled and disabled using ***gpio_pin_enable_callback(struct device *port, u32_t pin)*** and ***gpio_pin_disable_callback(struct device *port, u32_t pin)*** functions which basically generate a system call to set and unset the callback on port.

After all these initializations the device is set and ready to be used and also the APIs populated using the sensor_driver_api struct for the device can be used from the application to play with the sensor device. To get the device structure pointer of the sensor device created we will again use ***device_get_binding()*** and pass the argument as the configured device name (e.g. CONFIG_HCSR04_00 or CONFIG_HCSR04_01 in this assignment) to get the respective device structure pointer.



```
struct sensor_driver_api {
        sensor_attr_set_t attr_set;
        sensor_trigger_set_t trigger_set;
        sensor_sample_fetch_t sample_fetch;
        sensor_channel_get_t channel_get;
};
```

Fig 4. struct sensor_driver_api present in include/sensor.h [1]

The sensor device pointer obtained in the application can be used to call the APIs of the sensor to fetch and get the readings. The two APIs used for the HCSR04 sensor in this assignment are ***hcsr04_sample_fetch(struct device *dev, enum sensor_channel chan)*** and ***hcsr04_channel_get(struct device *dev, enum sensor_channel chan, struct sensor_value *val)***. These APIs can be called from the application using ***sensor_sample_fetch(struct device *dev)*** and ***sensor_channel_get(struct device *dev, enum sensor_channel chan, struct sensor_value *val)*** . Here the first argument in both is the sensor device structure pointer obtained for the sensor device and the other arguments for ***sensor_channel_get()*** are sensor channel on which the sensor should perform and the sensor_value struct exposed for the applications from the driver to pass the readings of the sensor. Both these APIs will do a system call and look for the device name amongst the sensors defined and then accordingly call their respective APIs to perform operation. The APIs to be used from the application are generic and easy to use and the specific sensor's APIs are not required to be known even if they have unconventional naming. These generic APIs will find their associative API from the sensor driver and then perform its functionality.

- **test_led program:**

In the test_led program given with assignment 1 the following devices are created/used:

1. **pinmux** – pinmux device structure.
2. **gpio_dw** – the Quark GIP GPIO Controller driver on board.

In test_led program the GPIO pin initialization routine is similar to the one explained above for the HCSR04 sensor driver. First the pinmux device is initialized and created and it's device structure pointer is obtained using device_get_binding() using the CONFIG_PINMUX_NAME for pinmux driver. Then the gpio_dw device is initialized and created and this device can be obtained from the pinmux driver data structure which contains all the drivers for the chips present on board. These two devices can then be used in a similar way as explained before to set the IO pins and the internal GPIO pins as per requirements of the application. In the test_led application after the pins are configured and set a callback is configured and added for GPIO pin 6. The program also gets the base address of the gpio_dw driver from its driver data which is a gpio_dw_runtime structure.

- **Assignment 1 program:**

In addition to this, in assignment 1 one more device was initialized and created for pwm_pca9685 driver for accessing the pwm internal pins present on board. These pins were set using ***pwm_pin_set_cycles(struct device *dev, u32_t pwm, u32_t period, u32_t pulse)*** API by providing the pwm device structure pointer obtained using using device_get_binding() with CONFIG_PWM_PCA9685_0_DEV_NAME as its argument as pwm driver name.

- **Assignment 2 application explanation:**

The implementation include 4 shell commands and 2 driver api and they work as follows :

1. During boot all the devices are disabled and the elapsed start time for the application is recorded for reference to show the elapse stop time for each sample measurements.
2. **'hcsr select n' –** For n=0, no devices are enabled. For n=1, only HCSR0 is enbaled. For n=2, only HCSR1 is enabled and for n=3, both HCSR0 and HCSR1 are enabled. For any other value no devices are enabled or disabled and the current devices which are enabled are preserved and the console prints a message saying n should be between 0 to 3.
3. **'hcsr start p' –** Max value of p is 256 and hence a buffer of size 512 is present in the application to accumulate all samples for both devices when both are enabled. Depending on the devices which are currently enabled, a sample fetch request is initiated p times, each followed by a sample get request for each device simultaneously. The received readings from the driver in centimeters are then stored one after the another along with driver name and elapse time in milliseconds since start of application. All new readings are overwritten and start filing the buffer from the start. For no devices enabled, it prints No measurements initiated as all devices disabled.
4. **'hcsr sump p1 p2' –** All the readings currently present in the buffer between p1 and p2 are dumped if p1 <= p2 or else it prints Value of P1 should be less than equal to P2.
5. **'hcsr clear' –** This clears the entire buffer.
6. **sensor_sample_fetch –** This triggers a new measurement if driver is not busy and generates a rising edge for the trigger pin and then waits for more than 10 microseconds and writes the trigger back to 0. The callback function is initiated here and takes the start time from rising edge and then stop time and distance measured at falling edge of the pulse received at echo pin and marks device free and give semaphore for channel get. The readings are stored in internal buffer of the device.
7. **sensor_channel_get –** It returns the distance measure and present in the internal per-device buffer. If buffer is empty, it blocks the channel using semaphore until new measure arrives and if the device is not busy but buffer is also empty then a new sample request is triggered. After each channel get, the internal per-device buffer is cleared.

References:

1. https://elixir.bootlin.com/zephyr/v1.14.1/source
2. https://docs.zephyrproject.org/1.14.1/introduction/index.html
3. CSE 522, Spring 2021 Lecture Slides by Dr. Yann-Hang Lee