CSE 522 - Real-time Embedded Systems - Spring 2021
Report - Thread programming and device driver in Zephyr RTOS

Shyam Joshi - 1218594676
sjoshi46@asu.edu

Device driver and it's initialization.

If you look at any device driver's (.c) file, there will be `DEVICE_AND_API_INIT()` or `DEVICE_INIT()` present at the end of the file. It creates a device object and sets it up during boot time initialization. Let's take a look at it's inside and figure out what it does. The below definition is from `device.h` present inside /zephyr/include/

```
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 will define and populate two static structures: struct device_config and struct device. device_config holds the configuration information for the device and struct device holds the driver data, pointer to it's APIs and pointer to the device_config structure.

Let's examine the arguments of the macro:

- *dev_name* is the name that will be assigned to the device structure used to represent the device. This length must be less than 48 bytes.
- *drv_name* is the name of the driver that will be assigned to the name member of the *device_config* structure.
- *init_fn* is the address to the init function which will be invoked during boot time initialization for setting up the device.
- *data* is the pointer to the variable where driver dependent data is stored for the device.
- *cfg_info* is the address to the structure which contains configuration information for the device.
- level is the level at which the init function is invoked. It must be one of the following: PRE_KERNEL_1, PRE_KERNEL_2, POST_KERNEL. If you want to use kernel services while initialization then your option is POST_KERNEL, else use the remaining two. PRE_KERNEL_1 is used when the driver solely relies on the hardware present in the processor/SoC and there are no kernel services available yet. PRE_KERNEL_2 is used for devices that rely on initialization of devices initialized as a part of the PRE_KERNEL_1 level. [from macro definition in device.h] PRE_KERNEL_1 and POST_KERNEL_2 runs on interrupt stack and POST_KERNEL runs in the process context of kernel main task.
- *prio* is the initialization priority of the device relative to the other priority of the same level. Value ranges from 0 to 99 with higher priority associated with lower values.
- *api* is the address to the structure holding the APIs for the device.

The below declaration is from `device.h` present inside /zephyr/include/

```
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;
};
```

```
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
};
```

Let's look at the implementation of the macro.

Regarding structure device_config,

1. *static struct device_config* with name equal to *dev_name* is created.
2. Inside attribute space, this structure is stored.
3. device_config's member, *name* is equated with *drv_name*.
4. device_config's function pointer, *init* is assigned to *init_fn*.
5. device_config's member, config_info is pointed to *cfg_info*.

Regarding structure device,

1. *static struct device* with name equal to *dev_name* is created.
2. Inside attribute space, this structure is stored with *init*'s priority assigned.
3. device's *config* pointer is assigned to the structure created above.
4. *driver_api* pointer is assigned to the *api* structure.
5. *driver_data* pointer is assigned to the *data* parameter.

`DEVICE_INIT()` expands as following:

```
#define DEVICE_INIT(dev_name, drv_name, init_fn, data, cfg_info, level,
prio) \
	DEVICE_AND_API_INIT(dev_name, drv_name, init_fn,\
	data, cfg_info, level, prio, NULL)
```

It calls `DEVICE_AND_API_INIT()` with the same argument except that for *api* it passes *NULL*.

There is another way to define a device which has an option to include device power management control i.e. `DEVICE_DEFINE()` which defines device_pm structure along with calling `DEVICE_AND_API_INIT()`.

The device objects are created and placed in memory by the linker. To retrieve any device, we use

```
struct device *device_get_binding(const char *name)
```

with parameter same as *drv_name* used in `DEVICE_AND_API_INIT()` It expands to the following as defined in /kernel/device.c

```
 struct device *z_impl_device_get_binding(const char *name)
{
      struct device *info;

      for (info = __device_init_start; info != __device_init_end;
info++) {
            if ((info->driver_api != NULL) &&
                (info->config->name == name)) {
                  return info;
            }
      }
      for (info = __device_init_start; info != __device_init_end;
info++) {
            if (info->driver_api == NULL) {
                  continue;
            }

            if (strcmp(name, info->config->name) == 0) {
                  return info;
            }
      }
return NULL;
}
```

It starts with the first device and access name of the driver through the *config* pointer stored in it and compares the same with the argument passed in the function. If there is a match, it returns the pointer to that device.

Suppose the driver_api structure for a device is defined and used as follows:

```
typedef int (*device_control)(struct device *dev);

int print_device_name(struct device *ptr)
{
  printk("Device Name = %s \n",ptr->config->name);
}

struct device_api {
      device_control control;
};
device_control.control=print_device_name;
```

The address of the API structure is used as a parameter during `DEVICE_AND_API_INIT()`

This API can be accessed from anywhere provided we know the name of the device as shown:

```
struct device *dev = device_get_binding(dev_name);
dev->api->control(dev);
=> dev_name
```

To acquire device parameters associated with the device, they must be defined in the *Kconfig* file as following:

config *parameter_name*
    data_type “information”
    default *value*
    help
        description about the variable.

data_type is the primitive data type (int, char, etc.)
value is the default value associated with the device.

Now to access this variable inside the driver file, simply equate any variable with CONFIG_PARAMAETER_NAME as shown below:

config HCSR0_ECHO
    int “Echo pin for HCSR0 sensor”
    default 3
    help
        Echo pin used for HCSR0 sensor

```
int variable = CONFIG_HCSR0_ECHO; printk("variable=%d \n",variable);
=> 3
```

List of devices used in the *test_led* program.

1. *static struct device *pinmux;*
2. *struct device *gpiob;*

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

The above line will acquire the pointer to the pinmux device stored in drivers. There are functions provided by the class driver of the device which can be accessed using this device.

pinmux has a member called *driver_data* which points to *galileo_data*'s structure. It contains the following:

```
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;
};
```

Now, the question is which pin multiplexer you want to control? Well the above devices listed inside galileo_data's structure are your available options. Galileo board has a bunch of devices from which the pin multiplexer can select the signal from.

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

gpio_dw is used to configure GPIO pins as input/output, add an interrupt, set/clear the value of the pin, etc.