Report Assignment 2: Thread Programming and Device driver in Zephyr RTOS

Arjun Bhatnagar
1215353243

Introduction

In assignment number 2 we are required to develop a HC-SR04 sensor driver in Zephyr RTOS 1.14.0 running on Galileo Gen 2. The driver was created as such it creates two instances of the sensor. Further we developed an application to take distance measures form either on the two sensors. The following parts of the report talk about how the objectives/requirements of the assignment were achieved and why.

1. Device driver model and Configuration options and Cmake file:

The Zephyr kernel provides various device drivers. The availability of a driver depends on the board and the driver. The zephyr model has a generic API for each device type. The way it works is that each driver gets a pointer to a structure which contains pointers to API functions. These structures are available in RAM placed according to initialization level. Our new sensor device driver works the same way.

1.1 Driver Data Structures:

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

struct device_config {
    char    *name;
    int (*init)(struct device *device);
    const void *config_info;
  [...]
};
```

Each device has a device struct, it is a per device instance containing device_config which is build time configuration information. The driver_api which is a pointer to the structure driver_api which maps our API functions (implementations) to the API subsystem. A pointer to driver data structure in RAM used to keep record of runtime information. The config member is for configuration data which to be set at build time i.e. the fixed data for the device passed through the config_info pointer (these are read only). The config member also contains a pointer to the name of the driver exposed to the system.

1.2 The API used to implement our driver:

`DEVICE_AND_API_INIT`(dev_name, drv_name, init_fn, data, cfg_info, level, prio, api)

This function creates our driver objects and the cause the driver initialization at boot time. It also links the driver API by taking a pointer to the API structure. The other parameters of this function include device name this name is looked up when we do device_get_binding. Argument drv_name is the name the device exposes to the system. So when we bind a device we retrieve a pointer to the device structure of the driver by the name it exposes to the system. Argument init_func passes the address of the initialization function, this function is called at boot (depending on initialization level). Then the pointer to the driver data containing runtime information is passed (runtime information). The cfg_info argument is used to pass a pointer to the device configuration information. We do not pass any address to cfg_info for pointing to build time information in our implementation. The initialization level of our diver would depend on the dependencies of the driver. Since our driver depends on GPIO device we set it up for POST KERNEL initialization. This allows the driver to use kernel services during its own initialization. We can set the priority level of the device in the range of 0 to 99 using prio. At last we pass the pointer to the API struct mapping the API subsystem.

Once we have created our device data structure containing run time information and implemented our API and initialization function we can communicate with our devices by binding the device instance by its name (device_get_binding("dev_name")). This passes the pointer to the device struct for our use. Using this pointer we can access the data objects of the driver and implement our functionalities through our API.

The important task of this assignment was to create two instances of the same driver to accommodate two devices. These instances share the same API but have different names and different gpio configuration information (pins). We need two instances of the driver data structures (driver_data and config_info). Hence we create two instances of the driver data (run time info) and call the DEVICE_INIT function with two different names which can be used to bind two different devices. Now we have two per device structures containing different information.

The way we pass information to configure these two instance is through the kconfig file of the driver. The kconfig file is a top level level file containing specific configuration parameters for the driver. Before we can use the config parameters we have to include the kconfig in zephyr by sourcing the driver kconfig in the overall sensor kconfig:

At drivers/sensor/hcsr/Kconfig
Included via Kconfig → Kconfig.zephyr → drivers/Kconfig → drivers/sensor/Kconfig (source here).

We also add the driver.c file to Cmakelist for it to be included in when building.

zephyr_library_sources_ifdef(CONFIG_HCSR04 hcsr04.c) and then add to the cmakelist for all the sensors.

In Kconfig we configured the HCSR driver with a menuconfig this allows us see its default setting as a menu (make menuconfig). The HCSR driver was set as a select statement (by using bool) forcing this config parameter in prj.conf (=y). This selects our driver for use. Then we set the names of the driver instances as strings and the cofig pins as integer values and access them as CONFIG_"name" in code.

2.Implementation of driver API:

To implement the sensor API we have sensor subsystem exposing it. Sensors can have multiple channels to represent different properties or different types of the same value. The results are returned as a values of a data structure struct sensor_value. This is done to avoid floating point representation of sensor results.

For fetching values: we implement a sample_sensor fetch to execute a distance measurement. We used an interrupt driven approach by binding a gpio device (on galileo) and using the gpio API in the driver code. The application first instructs to fetch a sample by calling sensor_sample_fetch using the pointer to the device struct we got while binding. Using the same pointer in our driver we get a pointer to its run time information for execution. The sample fetch call triggers the sensor using the trigger pin and then the sensor generates an interrupt at the echo pin which initiates a measurement routine to catch rising and falling edges. At the end of the routine if the measurement was successful we write the distance value in a per device runtime buffer and post a semaphore. Also while the measurement is being taken we have a status bit (also run time and per device) which is set to indicate an ongoing measurement (set to zero when a measurement is complete).

For getting values: we implement a sensor_channel_get to return distance values to user when a measure is received through fetch. The way it works is that if no value has been fetched we check if a measurement is underway. If yes we wait using a semaphore on the distance value, if this value is not available within the required time the device will time out and return from the measurement with a -1 value. If the the value is available it will return that. If no measurement is underway the measurement will start a new measurement and wait for the value.

For attribute set: using sensor attribute set the user can set the runtime attributes of the device (timeout value in our implementation).

3.Application to use the driver:

The application simply asks you to select a sensor. According to the selection it binds the device and then can start to measure the distance values. Running the start command starts the fetching and getting of sensor data values. Finally one can print these values by using the dump command.