PiTank HAL Software Architecture

Table of Contents

1. Hardware Abstraction Layer (HAL) Software Architecture

Approach 1 :

HAL based on Microchip Application Note here.

Below is the HAL project directory structure created based on above application note on raspberry pi.

pi@raspberrypi:~/Workspace $ tree -l

```
.
└── PiTank
    └── hal
        ├── app
        ├── bsp
        │   ├── app
        │   └── peripheral
        └── drivers
```

The *app* directory contains interfaces for Application layer. Application should only use api in app directory.

1 Hardware Abstraction Layer

Hardware Abstraction Layer (HAL) provides function API-based service to the higher-level layers

1.1 Hardware Access Function

In HAL access function provide access to low-level peripheral drivers. These function are therefore in *app* directory under *hal*

This module is a **Facade pattern**. Analogus to C++ pattern therefore, its like an abstract base class containing all behaviour (protected function, so it hints only 1 level deep inheritance) and derived classes implementing behaviour. Application program only to interface which is base class.

Similarly in C, above design goal is implemented by HAL by providing interface in *app* directory which will provide access to driver functions. To see example code, see same Microchip App note here.

We follow therefore following design principle (Which are generic, used above in Facade, used in Strategy too)

design principle 1 : Identify the aspect of your application that vary and separate them from what stays the same.

Seperate non-varying part and make it interface for application. Seperate variying or behaviour classes and use pointers to them (*has A*) in non -varying.

principle 2: Program to an interface not implementation.

An example of above principles is here.

1.2 Board Support Package

The primary objective of the Board Support Package (BSP) module is to provide one access point for users to modify the hardware mapping details without requiring an extensive search and replace operation. In order to achieve this objective, the BSP is divided into two inter-dependent interfaces:

Application Interface and *Peripheral Interface*.

This type of an approach localizes the behavioral / functional mapping to the *Application Interface* while keeping the actual hardware mapping within the *Peripheral Interface*.

Application interface maps to *Peripheral interface* and *Periferal interface* to *Drivers*. Follow Microchip App Note for examples.

On raspberry code project, Application Interface is therefore in *app* directory and Peripheral Interface in *peripheral* directory.

1.3 Drivers

Peripheral drives access peripherals directory. In raspberry project, source is under *drivers* directory.

Approach 2 :

2 HAL & BSP development

Above definition for HAL and BSP are from Microchip Application Notes and are applicable for their MPLAB IDE and Harmony framework. But In general

BSP : In embedded systems, a board support package (BSP) is the layer of software containing hardware-specific drivers and other routines that allow a particular operating system (traditionally a real-time operating system, or RTOS) to function in a particular hardware environment (a computer or CPU card), integrated with the RTOS itself. Third-party hardware developers who wish to support a particular RTOS must create a BSP that allows that RTOS to run on their platform.

Its purpose is to provide the board support required by a specific OS. So for the same hardware, different OS will likely require different BSPs.

As per this document from VxWorks, A BSP:

- A BSP is the kernel's interface to device drivers
- A BSP is a set of libraries offering a hardware abstraction layer to kernel
- A BSP supports a host/target cross development environment

Now above points are applicable to a RTOS, and we have Linux – a monolithic kernel on Raspberry. On Raspberry Pi, therefore we already have board support e.g gpio, spi etc drivers, and packages like WiringPi further provide userspace drivers based on kernel space drivers exposed by sysfs.

So a BSP is for a particular OS.

HAL : HAL on the other hand, as name suggests is an abstraction Layer. It is a common interace

that is used by application. So HAL interface remain same and its implementation is different for different hardware, while underlying BSP and OS can be changed.

As described above, HAL is therfeore facade. As an example, it can be implemented as discussed in this StackOverflow thread

1. Define an API for each component in the HAL, (typically structs of pointers-to-function, such as *adc_driver_api_t, uart_driver_api_t, i2c_driver_api_t* etc.) These APIs do not use any chip/compiler extensions or includes, but typically adhere to a language standard (C89/C99).
2. The HAL for a given platform provides implementations for these, e.g. *msp430_adc12_driver, msp430_adc10_driver, pic18_adc12_driver, avr_adc_driver* etc all implements the *adc_driver_api_t* interface using different peripherals available on the respective platforms. Each implementation expose a const global instance of the implemented driver in the implementation header file, e.g. (using C) *extern const adc_driver_api_t msp430_adc12_driver*;. These implementations use chip/compiler extensions or includes as required.
3. A component using the HAL to get ADC readings would be initialised with a const *adc_driver_api_t* * implementation and anything else the API requires (perhaps an adc channel). The component and the ADC driver implementation would both be initialised and connected together in the program initialisation, e.g. top of main()

3 HAL Design on Raspberry Pi

With Approach 2 in mind, following directory structure is created on raspberry pi board

```
└── src
    ├── app
    │   ├── motor_control.c
    │   └── motor_control.h
    └── hal
        └── pi-linux
            ├── bsp
            │   ├── bsp.c
            │   └── bsp.h
            ├── drivers
            │   └── pwm
            │       ├── pwm.c
            │       └── pwm.h
            └── motor_control_hw.c
```

HAL implementation under *pi-linux* directory contains all drivers and BSP for raspberry-pi board with linux OS.

On Application side, in directory app, *motor_control.h* specify common interface that is expected from any HAL. This interface is implemented by particular HAL implentation under *pi-linux* directory in *motor_control_hw.c*. pi-linux directory contains code for hal implementation for raspberry-pi board with linux OS. Under *bsp* directory code contains hw mapping and board specefic macros. Since OS is Linux which is not a RTOS, most of the drivers are already available. So drivers here contains mostly userspace drivers based on interface exported by sysfs.

4 Development on Raspberry Pi board

Raspberry Pi 3 Model B Ver 1.2 is used in this project.

To drive ESC for motor control ie L298 motor driver, PWM will be generated on Pi and output on Pi GPIO.

Board Hardware file are available on Pi website at Link

DOCUMENTATION > HARDWARE > RASPBERRYPI

Here you find Schematics, GPIO documents, and GPIO usage.

On Raspberry Pi, all GPIO banks are supplied from **3.3V**. Connection of a GPIO to a voltage higher than 3.3V will likely destroy the GPIO block within the SoC.

Since Hardware PWM is available on RaspberryPi, We can use HW PWM output pin e.g GPIO18 and WiringPi library for API.