Chimera Embedded Driver Architecture

Brandon Braun

October 6, 2020

Contents

Abstract

The Chimera Embedded System Driver architecture is split into high level and low level drivers to facilitate abstraction of hardware level MCU peripherals into a common framework. This eases development of new peripheral interfaces and provides a unified layout for projects to build on. This document summarizes the structure of the drivers and various thoughts behind its design.

1 Introduction

The goal of Chimera is to provide a standard C++ API upon which I could build all my future embedded projects upon. While Linux exists and is a very useful and widely accepted approach to hardware driver development, it didn't offer any opportunity for personal growth in software architectural design. While a labor of love, Chimera is slowly improving into a useful tool for accessing common embedded peripherals in a manner that takes into account modern C++ concepts. The idea is that this approach will drive faster development of future projects by simplifying the requirements to get hardware up and running, while also offering more complex and modern functionality.

From a conceptual perspective, Chimera provides an architectural specification that outlines how drivers for custom CPU/MCUs should be created, which is split into a high level driver(HLD) and low level driver (LLD). The HLD is platform agnostic and is responsible for the 'expected-use' functionality of a particular peripheral. Typically this involves generic configuration settings, reading and writing data, handling event callbacks, etc. The LLD is platform specific and is responsible for implementing hardware configuration aspects of the peripheral. It translates the generic HLD peripheral settings into the proper bits, bytes, and words that enable the desired functionality in the hardware. By following these specs, software built on top of it can be easily ported to other devices without much hassle. Simply swap out the low level driver and a new device is ready to go.

For reference, this document outlines the Thor driver implementation for Chimera, which adds STM32 device support. While some of the implementation details in the Thor LLD might not be portable to other devices, it serves as a good example of the kind of detail that might be needed to implement Chimera on another device.

2 Thor High Level Driver (HLD)

2.1 Introduction

The Thor high level driver handles commonly desired abstractions for various peripheral interfaces to help ease the integration of hardware peripherals into application code. The high level driver conforms to the Chimera interface specification through the use of pure virtual inheritance to ensure that applications can safely build on top of a Chimera driver implementation without worrying about the hardware details.

2.2 Generic Structure

Each peripheral driver follows a particular structure to provide a consistent layout of the driver software. By keeping the folder/file structure consistent, it is expected to allow easier navigation of other peripheral drivers in the system. The current structure is outlined below and in later sections is described in detail:

```
HLD Peripheral Driver (GPIO Example)
├── test
│   ├── test_entry_gpio.cpp
│   ├── test_fixture_gpio.cpp
│   ├── test_fixture_gpio.hpp
│   ├── test_hld_gpio_chimera.cpp
│   └── test_hld_gpio_driver.cpp
├── build.jam
├── hld_gpio_chimera.cpp
├── hld_gpio_chimera.hpp
├── hld_gpio_driver.cpp
├── hld_gpio_driver.hpp
└── hld_gpio_types.cpp
```

hld_<*peripheral*>_chimera.hpp:
Publicly declares the driver's interface to Chimera. It is expected that this will only be used internally within the driver implementation, but it may be useful to include this elsewhere.

hld_<*peripheral*>_chimera.cpp:
Implements the driver registration functionality to Chimera. Typically this involves definitions of the function prototypes in the associated driver interface header file and a function(s) that register the driver with the Chimera back end. If no driver is currently implemented for the given peripheral, then default behavior can be provided here.

hld_<*peripheral*>_driver.(c,h)pp:
These two files define the full HLD interface that can be used by projects. At a bare minimum, a peripheral driver class is declared that inherits from the Chimera virtual base class for that peripheral. This will guarantee the proper interface spec is upheld. Additional functionality can be declared that is not defined in Chimera, but it will not be propagated to Chimera peripheral objects.

hld_<*peripheral*>_types.hpp:
Defines commonly used types that are shared across all driver specific implementations. There is not likely to be much in this file as it's common for the Chimera driver defintions to be sufficient.

2.3 Build System

2.4 Chimera Hooks

2.5 Driver Implementation

2.6 Test Implementation

3 Thor Low Level Driver (LLD)

3.1 Introduction

3.2 Interface Layer

3.3 Driver Layer

3.3.1 Generic Structure

3.3.2 Build System

3.3.3 Example Driver Implementation (CAN)

A hardware driver can be broken up into three main components: Primary logic, chip variant definitions, and system tests. Each of these components serve a specific purpose and are broken down as follows.

Primary Logic
This section forms the core of the hardware driver and serve to implement register level interactions as well as satisfy requirements from the interface layer. This is achieved through the use of several kinds of files.

The low level driver interface is satisfied by *hw_can_driver.cpp*, which houses the definition of the core driver class declaration. This fully encapsulates all the idiosyncracies that may be present in the actual hardware. Note that the interface does not specify thread safety at this layer, so where possible, the driver should be atomic. In general though, it's safe to assume that under proper use cases, the driver will be called under protection from higher level driver thread safety implementations.

Additionally, data requirements from the interface layer are satisfied by *hw_can_data.cpp*. This contains configuration mapping structures, lists of available peripherals, options supported by the driver, etc. Most of these data fields will be constant as they indicate hardware properties that should not change, either because they are set on boot or they are hardwired into the silicon.

Not all devices are the same, so definitions are collected into *hw_can_prj.hpp*. This file uses compiler flags to select the proper header file for a given chip variant from the Variant folder. At a bare minimum, it includes the top level common definitions, then selectively includes additional definitions based on the chip.

Finally, common types that describe hardware characteristics, behaviors, configuration options, etc are described inside of *hw_can_types.hpp*. These should *not* be used in higher layer drivers if possible as it introduces tight coupling between the HLD and LLD.

Chip Variant Definitions
These files describe the physical hardware and all configuration options. It is responsible for two levels of description, a high level version that contains definitions common to all variants, and a lower level version that is highly chip specific. For example, within a device family, a hardware peripheral typically shares the same core register types, configuration bits, and fields. Some chips may have a bit here or there turned on/off, but by and large they are the same. Alternatively, not all chips have the same number of peripherals, or some core features are disabled, etc. This kind of descriptive information would go into the aforementioned lower level version.

The structure of these files are contained as such: The high level definitions go into *hw_can_register_stm32l4xxxx.hpp* and the low level definitions into *hw_can_register_stm32l432kc.(c,h)pp*

System Tests
The test drivers for each peripheral are intended to interrogate the low level driver on the physical device. As such its scope is fairly limited and should not extend to behavior outside of the LLD interface layer described previously. Each peripheral will have a core set of tests inside of *test_lld_can_driver.cpp* and any additional tests inside of *test_lld_can_xyz.cpp*. These tests are likely to be small as they will be constrained by the on-chip Flash and RAM.