0 Foundation Framework

There are two conceptions must be distinguished before, which are platform and framework.

The **platform**, on which we work. It determines what we can do.

And yet, the **framework**, which will guide and restrict our work. It determines how we might do.

0.1 Why to do

0.1.1 Project Import Easily

Many people feel that it is painful to get start of a work. So is the project of development and researching. This problem entangles either developers in application layer or programmers of BSP and device driver developing.

So many people need a guide to introduce how to kick off a work. The foundation framework, as guide, will help programmers working easily without time exhausting in kernel detail. Even though, we still describe and deduce kernel in detail.

0.1.2 Debug Convenient

The other problem is that it is hard to trace program running and interaction with kernel, especially when developing device driver.

Now CPU original manufactories, like ARM with Keil, usually have excellent developing tools chain and provide powerful debug tool for us. Yet we can't use their convenient in Linux development as well for unknown reason.

0.1.3 Device Driver Migration

We find that many famous chip render, such as Boardcom, Marvel and so on, only provide Linux/Windows driver program for their powerful driving chips. More than that they only provide a little documents and notes with their program.

Thanks for Linux, we have chance to learn source code of the chip's driver program. By these source codes, we can get some undocumented information about work mode configuration, time sequential diagram, and operating method. Furthermore, we hope we can migrate its Linux driver with a little modification in our embedded system.

Thus we prepare a Linux middleware so that the chip's driver can be debugged step by step in a Linux similar environment.

0.2 What to do

Obviously, it is ultimate purpose to learn and debug device driver step by step in a Linux similar environment. Especially we can develop and debug Linux middleware with CPU original manufactory's IDE (compiler and JTAG debug tool) in windows.

Perhaps it is going to break a lot of Linux loyal hearts. Because the Linux beginner no longer need those inconvenient Linux development environment to learn and debug source code of Linux.

The other benefits belong to incidental small goal with this framework. Such as,

- Lightweight micro kernel
- Learning Linux kernel conveniently for beginner
- Kick off a new project easily
- Power managing clearly for low power application
- Hierarchical architecture make your project more ordered as protocol stack.

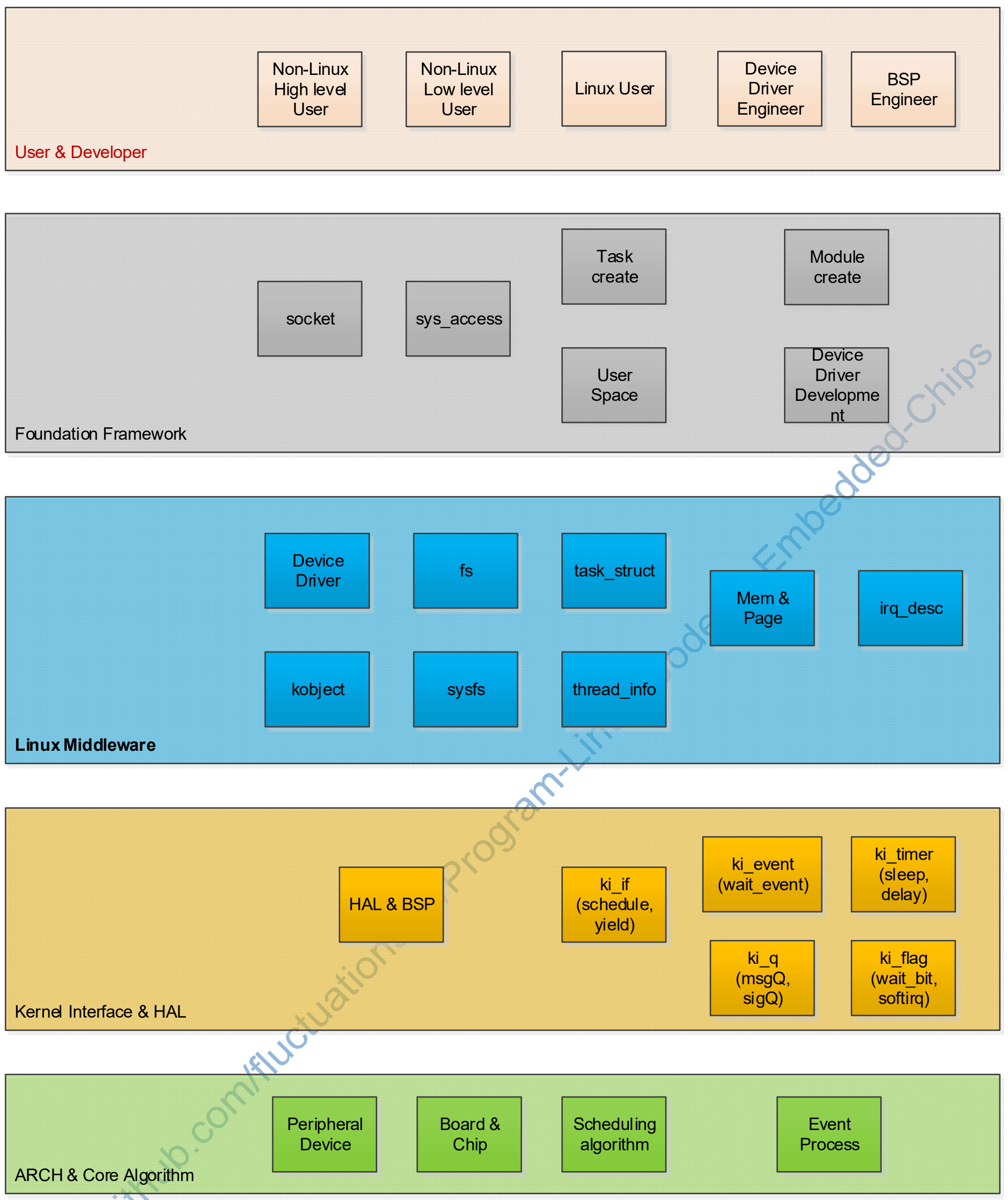
0.3 How to do

First of all, we should reorganize Linux system before deriving our Linux similar platform. We look at the skeleton of our system outlined as following,

The entire structure is divided into 5 layers. The top and bottom are unfixed parts.

The top layer represents either user or developer, who are all going to do something on this platform. We only concentrate on aspect of developing, including device driver and application. The developer perhaps design for new requirements or develop driver for new devices. They need a foundation framework to guide them kicking off a projects.

The bottom layer covers chip, circuit board and core algorithm about scheduling, memory and event processing. They have a common characteristic that is changeable according to cost, performance and purpose.



Both of msgQ and sigQ permit that there are more than one signal he message with the same ID can be stored in queue simultaneously. We distinguish them with distinctive parameter.
msgQ – with distinctive parameter
sigQ – with distinctive timestamp

Figure 0- 1 Skeleton view of the kernel

In this architecture, the middle 3 layers are most concerned for us. The upper layer of them is framework, supporting development of many kinds of application, device driver and BSP. The middle layer is Linux middleware which almost provides Linux API for every programmers of all layers including application, device driver and BSP. The lower layer is kernel interface which provide conversion interface between Linux kernel and scheduling and event processing algorithm. These algorithms can be realized by other RTOS core rather than Linux.

In the architecture of our expected, most of the kernel is still programmed in Linux style but based on micro kernel architecture. It means not only we can replace micro-chip, but also can change core of kernel such as scheduling algorithm, event and synchronous method. But we maintain majority of Linux data structure of kernel and system access interface. So it is still a developing framework of Linux's features to all programmers except some people who building micro-kernel interface.

Obviously, we get our prime goal, many Linux native drivers can running on this platform with a little adaption. Because this platform realizes the Linux middleware, and almost every driver just interaction with the middleware rather than thread's scheduling and synchronizing algorithm.

In addition to middleware, it is interesting to introduce the higher layer and lower layer.

The higher layer is our promoted foundation framework for development. Certainly we has not ability to realize automatic code generation. But we can provide a general programming pattern for some kinds of project. Mainly we will focus on device driver development, music effector and WPAN communicating application.

The lower layer consists of kernel interface (KI) and hardware abstract layer (HAL). These will help us to get the capability to alternate OS core (scheduling algorithm and synchronizing mechanism) and CPU. Usually we can the scheduling algorithm and synchronizing mechanism as *core of kernel*. This term will often appear in this book.

Following chapter, we will introduce how to build a device driver for sound card with SPI port in detail, and represent how to define a board resource.

Anyway, we will follow the principles below,

- **We do NOT construct and develop the Linux kernel, or Linux middleware in our scenario.** We only focus on edges and leaves associated with chip, circuit board and core algorithms. It perhaps take readers some confusion to consider the scheduling algorithm as edge and leave.

  Because we have a different perspective, which will focus on deconstructing instead of constructing a system. Thus the some algorithms, such as thread scheduling, memory allocating and event processing, are not the core part of our jigsaw, because they are all replaceable. **Only irreplaceable elements are able to become core.**

- **Employ MMR data structure of original manufactory as possible**

- **Employ HAL packages of original manufactory, such as STM, TI and ADI.**

- **Employ tool chains of original manufactory, such as STM, TI and ADI.**

  Although we can access chip and devices with register read/write directly. But it isn't economical and perhaps bring us any potential risks. Some chips, like TI's cc13x0 and STM32Fxx, have a few of private requests. For example, when we changing system clock, the two chips have distinct operating details in timing and logical relationship. It will result in failure when we access register directly. Especially in TI chip, it is certainly necessary to invoke a procedure preloaded in ROM to change system clock.

  The above 3 items will cause the following requests.

- **High Linux similarity**
  We don't make any modification until we encounter the following cases,
  1. **To abandon redundant procedures for multi mechanism**. As well know, the design thought of Linux provides multiple mechanism, not policy. Multi-mechanism will provides more than one approach to get oneself wanted. Actually it is very unfriendly to amateur because they might select hardly which one is better. Also multi-mechanism come with redundant procedure. Thus in this framework, a proper policy is almost the only choice, which absolutely is unity, distinct and understandable.

  2. **To delete symmetric multiple processors (SMP) code**. Our framework orients to embedded system, which is either single core or asymmetric 2 core. Thus, the spinlock and rw_spinlock are unnecessary and can be bypassed.

  3. **To avoid wait/complete mechanism when sysfs access.** Because in non-smp process, it is impossible to encounter confliction of sysfs access.

  4. **GNC/GCC particular compiler instructions.** In current phase, we are not going to support GNC/GCC compiler, but also insist on building projects with tool chain of only original manufactory in windows circumstance.

- Keep extendibility in the future, such as multi-person cooperative developing and remote procedure call