# The Analysis of PIN CONTROL Subsystem

Nadya A. Mohamed and Xu Chen
ECE Department, Rice University

## Abstract

In this report, we focus on a specific Linux subsystem - the PIN CONTROL subsystem, which is a centralized mechanism that can arrange and configure pins. We discuss the hardware background of the PIN CONTROL subsystem, the reason why it is necessary, and its three major functions. Finally, we perform our own analysis for this subsystem. We leverage cloc and cscope as analyzers, and observe several interesting facts about PIN CONTROL subsystem through the analysis.

## 1. Introduction

## 2. Pins in Hardware

### 2.1 Overview

What is pin?

### 2.2 SoC Pins

What is pin?

### 2.3 Pad Cell

What is pin?

### 2.4 GPIO

How GPIO block is formed using I/O Cell.

## 3. PIN CONTROL Subsystem

A system-on-chip (SOC) has plenty of pins on it. To ensure that a SOC work can normally, all pins should be properly configured. However, managing such complex configuration is quite challenging since the correct pin configuration is board-specific, and hence lots of duplicate codes are required in the pin configuration [2].

The PIN CONTROL subsystem, as a centralized mechanism, has three major functions that help address these challenges, pin enumerating, pin multiplexing (e.g., the way to reuse a single pin or a single group of pins for different purposes), and pin configuration [3].

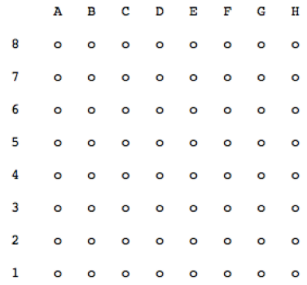



Figure 1: A single PGA chip [3]

### 3.1 Pin Enumerating

An important function of PIN CONTROL subsystem is enumerating, which means that it can enumerate all controllable pins that a specific processor provides. In the example of a PGA chip board seen from the underneath, as shown in the Fig. 1, we observe that PIN CONTROL subsystem is able to register a pin controller and name all pins by associating them with integers [3]. These integers are called pin numbers. Fig. 2 displays a typical industrial standard to name all controllable pins in a given board. Specifically, the pins A8, B8, C8 are named as 0, 1, 2 in the beginning, and the pins F1, G1, H1 are denoted as 61, 62, 63 in the end.

It is important to note that this naming strategy is only a typical example. Other naming principles are also allowed. The association between pins and pin numbers can be regarded as a bridge that connects the hardware and software, hence making further analysis from the perspective of software much easier.

In addition, controllers usually deal with groups of pins when performing functions, instead of a single pin. Therefore, a mechanism that can enumerate and retrieve ”groups of pins” is highly desirable [2]. These groups of pins are defined as the pin groups, which often correspond to a specific function [3]. For example, we can have one groups of pins 0, 8, 16, 24 (i.e., pin numbers) dealing with SPI interface, and the other groups of pins 24, 25 dealing with I2C interface. Once the pin groups are clearly defined, then the function in


ISBN [to be supplied]. . . $15.00

```
#include <linux/pinctrl/pinctrl.h>

const struct pinctrl_pin_desc foo_pins[] = {
      PINCTRL_PIN(0, "A8"),
      PINCTRL_PIN(1, "B8"),
      PINCTRL_PIN(2, "C8"),
      ...
      PINCTRL_PIN(61, "F1"),
      PINCTRL_PIN(62, "G1"),
      PINCTRL_PIN(63, "H1"),
};

static struct pinctrl_desc foo_desc = {
        .name = "foo",
        .pins = foo_pins,
        .npins = ARRAY_SIZE(foo_pins),
        .owner = THIS_MODULE,
};

int __init foo_probe(void)
{
        int error;

        struct pinctrl_dev *pctl;

        error = pinctrl_register_and_init(&foo_desc, <PARENT>, NULL, &pctl);
        if (error)
                return error;

        return pinctrl_enable(pctl);
}
```

Figure 2: Enumerating pins [3]

PINCTRL code get_group_count will be able to determine the total number of legal selectors needed for other functions to retrieve the names and pins of each pin group [3].

### 3.2 PINMUX

Given a system-on-chip, there are multiple ways to utilize its pins. However, a common situation that users might encounter is that some pins need to be reused for different tasks. To this end, a channel that can regulate the way to reuse pins for multiple mutually exclusive functions becomes necessary. PIN CONTROL subsystem has the PINMUX functionality, which helps address this issue.

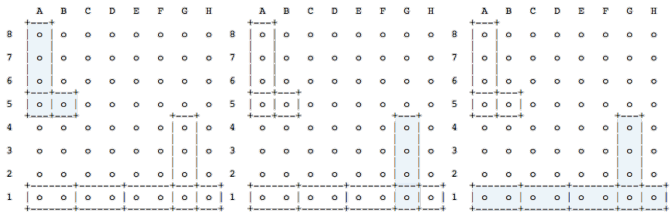
Fig. 3 shows a example of how the PINMUX mechanism works for a single chip. This example again utilizes a single PGA chip seen from the underneath [3]. When this chip is used, some pins will be taken by feeding power to the chip. They will be connected to VCC or GND. Another groups of pins will be occupied by large ports, such as the external memory interface [2]. After then, the remaining pins on the board will be subject to pin multiplexing (i.e., PINMUX) [3].

In the case where the pins A8, A7, A6, A5 are used to deal with the SPI port, and the pins A5, B5 are exploited to deal with I2C port, we observe that the A5 pin is actually "shared" by two ports. Due to this sharing, SPI and I2C ports could not be used at the same time. However, PINMUX settings enable the SPI logic to be routed out on the pins G4, G3, G2, G1, such that the conflict at A5 can be eliminated, and the SPI, I2C ports then can be exploited simultaneously. Similarly, while a 8-bit MMC bus is being used, the routed SPI port then will not be allowed to be utilized at the same time. PINMUX settings control all these processes [3].

### 3.3 Pin Configuration

The last function of the PINCTRL subsystem is pin configuration. The key idea of pin configuration is that pins can be software-configured since pins also have electronic properties [3].

Specifically, in the Fig. 4, once the switch in the upper of this circuit is closed, an input pin will be attached to VDD to




(a) Pins for SPI and I2C (b) Routed pins for SPI (c) SPI and MMC (8-bit)

Figure 3: Example of PINMUX use [3]

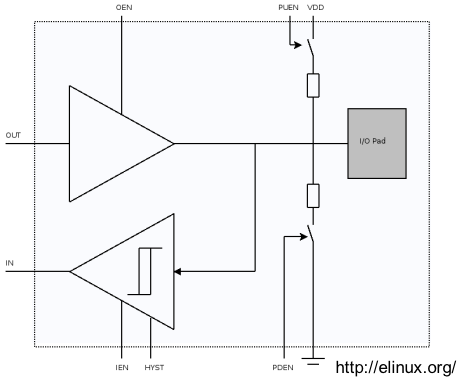



Figure 4: The illustration of pin configuration

provide power. We also notice that there is a pull-up resistor, which ensures that the input signal will be maintained at a stable level even when pins are disconnected, or when output pins are connected to a high impedance [5]. The lower part of this circuit contains a pull-down resistor, which serves a similar purpose, keeping the signal at low level when the pin is unconnected [6]. Pin configuration codes integrated in the PINCTRL subsystem is able to help set all these electronic operations of pins (i.e., hardware operation) from the software perspective [3].

## 4. PINCTRL Subsystem Analysis

### 4.1 Line of Codes (LOC) Analysis

To start with, we first discuss our findings in the lines of code (LOC) analysis. Fig. 5 displays the number of physical code lines, comment lines, and blank lines in the PINCTRL codes designed for multiple hardware in the version 4.11, the latest version we study. We pick up the several typical hardware examples that PINCTRL codes can work for, including ADI GPIO2 controller on bf54x processor (i.e., adi2-bf54) [7], GPIO driver for AMD (amd) [8], the power management unit AS3722 (as3722) [9], Atmel AT91SAM SoC (at91) [10], ST-Ericsson U300 Series (coh901) [11], Lantiq SOC (lantiq) [12], and PIC32 pin controller driver (pic32) [3]. In Fig. 5, the major observation is that most PINCTRL codes do have much shorter comment lines compared to the physical lines of code (LOC). We envision that it is partly since the

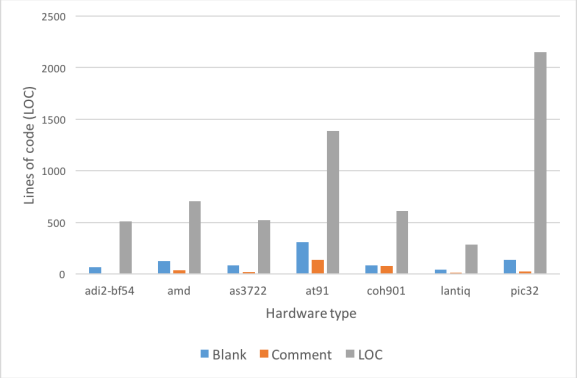
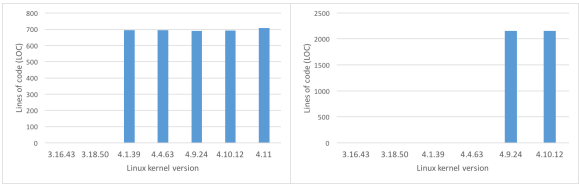



Figure 5: LOC of PINCTRL for various hardware v.4.11




(a) PINCTRL for hardware amd (b) PINCTRL for hardware pic32

Figure 6: LOC revolution of PINCTRL for two hardware

documentation [3] is comprehensive enough such that only a few of comment lines are in need to highlight the hardware-specific requirements.

As shown in Fig. 6, we also study the temporal development of the LOC in PINCTRL codes. According to the official document [4], the overall PINCTRL subsystem was added to the Linux kernel from the version 3.13. The codes designed for GPIO driver of AMD, and PIC32 pin controller driver, were not developed until the version 4.1.39 and 4.9.24, respectively [4]. But after they were introduced, these two codes have only experienced minor modifications in the past few versions.

Then we extend our analysis to the version-by-version difference of LOC in PINCTRL codes for different hardware. As shown in Fig. 7, we note that the PINCTRL codes for ADI GPIO2 controller and Lantiq SOC experience no change starting from their released dates. That means codes designed for them are comparably robust for long-term use. On the other hand, PINCTRL codes designed for Atmel AT91SAM SoC have been extended a lot from 3.16 to 3.18, and are shrinked from the version 4.1.39 to 4.9.24, which indicates the features in the initial version might not meet user demands. Finally, the PINCTRL subsystem for the power management unit AS3722 and ST-Ericsson U300 Series both have experienced minor changes from one version to another.

The above hardware's PINCTRL codes show that starting from the Linux kernel version 3.13 where the PINCTRL

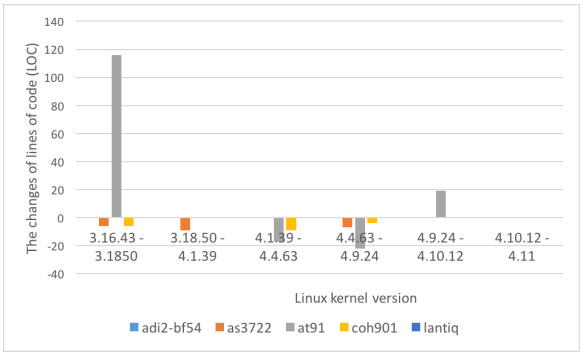



Figure 7: LOC difference across different versions

subsystem is initially introduced, the large-scale revisions of PINCTRL codes do not happen very often. Thus, generally the PINCTRL subsystem is stable and robust in the long-term use.

### 4.2 Author Database Analysis

### 4.3 Function Use Analysis

### 4.4 Methods, Tools, and Procedures

To start with, we download all source codes of Linux Kernel after the version 3.16.43, from the website [1] to perform our analysis. It is because the PINCTRL subsystem is introduced after the version 3.13. The PINCTRL codes are stored in the directory "Linux/drivers/pinctrl".

In LOC analysis, we leverage the tool **cloc** [13]. This toolkit can display the lines of code (LOC) for all the files in a single folder. It can deal with the files written in different languages (e.g., C, C++, HTML, XML, etc).

After installing this toolkit using Homebrew, we then place the PINCTRL target codes we aim to analyze (i.e., codes for different hardware) to different folders. The use of toolkit is based on command lines in Mac OSX Terminal. Using command lines, we first reach different folders that store the PINCTRL codes for different hardware, and then call **cloc** to detect physical lines, comment lines and blank lines for all files in different folders.

Moreover, this tool has several interesting functions, such as removing comments from source code, and analyzing codes within compressed archives. We also play with these functions during this project.

## 5. Conclusion

In this report, we study the PINCTRL subsystem extensively by examining its background information (i.e., hardware) and its three major functions. We then perform our own analysis for the codes of PINCTRL subsystem, using cloc and cscope tools. Our analysis involves lines of code (LOC) in multiple hardware, and xxx

## References


[1] The Linux Kernel Archives, https://www.kernel.org/

[2] The pin control subsystem, https://lwn.net/Articles/468759/

[3] PINCTRL subsystem https://www.kernel.org/doc/Documentation/pinctrl.txt

[4] Linux Cross References, http://lxr.free-electrons.com/source/drivers/pinctrl/

[5] Pull-up resistor, https://en.wikipedia.org/wiki/Pull-up_resistor

[6] Pull-down resistor, http://playground.arduino.cc/CommonTopics/PullUpDownResistor

[7] Pinctrl: ADI PIN control driver for the GPIO controller on bf54x and bf60x, https://lwn.net/Articles/565586/

[8] AMD GPIO Driver http://drivers.softpedia.com/downloadTag/AMD%20GPIO%20Driver
http://drivers.softpedia.com/downloadTag/AMD%20GPIO%20Driver

[9] AS3722 Power Management Unit http://ams.com/eng/Products/Power-Management/Power-Management-Units/AS3722

[10] AT91SAM Community, http://www.at91.com/

[11] ST-Ericsson U300 Device Tree Bindings https://www.kernel.org/doc/Documentation/devicetree/bindings/arm/ste-u300.txt

[12] Lantiq SOCs https://wiki.openwrt.org/doc/hardware/soc/soc.lantiq

[13] Count Lines of Code, http://cloc.sourceforge.net/