Preparation of Frame Work for the Customization of Device Driver in Linux Environment

P. Anusha
Assistant Professor
Department of Electrical and Communication Engineering
K G Reddy College of Engineering & Technology, Moinabad, RR District, Telangana

Abstract: The Linux kernel now has a coherent and uniform model to organize busses, drivers and devices. Device drivers must register themselves to the core kernel and implement a set of operations specific to their type: i) Character drivers must instantiate and register a cdev structure and implement file operations, ii) Block drivers must instantiate and register a gendisk structure and implement block device operations and a special make request function, iii) Network drivers must instantiate and register a net device structure and implement net device ops. The organization of device drivers has been greatly customized and unified by using this frame work preparation (model). This model and the Linux kernel in general, use some concept of object-oriented programming to structure the code. Functionalities such as udev have been made possible using this unified model. This paper describes how a device can be customized on the architecture OMAP4460 (ARM cortex series).

Keywords: Kernel, Device driver, Framework, OMAP4460.

I. INTRODUCTION

The Kernel-Mode Driver Framework (KMDF) model continues to allow development of kernel-mode device drivers, but attempts to provide standard implementations of functions that are known to cause problems, including cancellation of I/O operations, power management, and plug and play device support. *Device drivers* are loadable kernel modules that manage data transfers while insulating the rest of the kernel from the device hardware. To be compatible with the operating environment, device drivers need to be able to accommodate such features as multithreading, virtual memory addressing, and both 32–bit and 64–bit operation.
The kernel provides access to device drivers through:

- *Device-to-driver mapping:* The kernel maintains the *device tree*. Each node in the tree represents a virtual or a physical device. The kernel binds each node to a driver by matching the device node name with the set of drivers installed in the system. The device is made accessible to applications only if there is a driver binding.
- *DDI/DDK interfaces* — DDI/DDK stands for Device Driver Interface Driver-Kernel Interface. The DDI/DKI interfaces standardize interactions between the driver and the kernel, the device hardware, and the boot/configuration software. Use of these interfaces keeps the driver independent from the kernel and improves its portability across successive releases of the operating environments on a particular machine.

1. Device driver

A *device driver* is a kernel module responsible for managing low-level I/O operations for a particular hardware device. Device drivers can also be software-only, emulating a device that exists only in software, such as a RAM disk or a pseudo-terminal. A device driver contains all the device-specific code necessary to communicate with a device and provides a standard set of interfaces to the rest of the system. This interface protects the kernel from device specifics just as the system call interface protects application programs from platform specifics. Application programs and the rest of the kernel need little (if any) device-specific code to address the device. In this way, device drivers make the system more portable and easier to maintain. Auto configuration is the process of getting the driver's code and static data loaded into memory and registered with the system. It also involves configuring (attaching) individual device instances that are controlled by the driver.
Configuring the device procedure is based on the following:
_ "Driver Loading and Unloading"

_ “Data Structures Required for Drivers”
_ “Loadable Driver Interfaces”
_ “Device Configuration Concepts”
_ “Using Device IDs”

Device drivers typically include the following:
_ Device-loadable driver section
_ Device configuration section
_ Character driver entry points

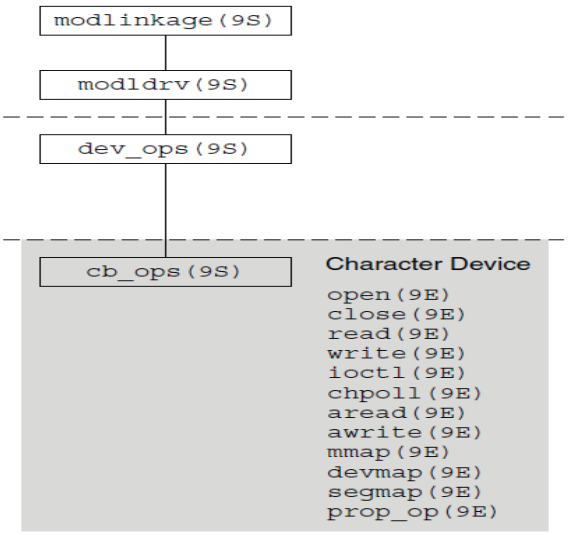

Figure 1: Character Driver Roadmap

Associated with each device driver is a dev_ops() structure, which in turn refers to a cb_ops() structure. These structures contain pointers to the driver entry points. Note that some of these entry points can be replaced with nodev() or nulldev() as appropriate.

2. OMAP4460

The dual Cortex™-A9 microprocessor unit (MPU) subsystem of the device is based on the symmetric multiprocessor (SMP) architecture. Thus, the dual Cortex-A9 MPU subsystem delivers higher performance, optimal power management, and debug and emulation capabilities. The dual Cortex-A9 MPU subsystem incorporates two Cortex-A9 central processing units (CPUs), level 2 (L2) cache shared between the two CPUs, and uses PL310 as an L2 cache controller. Each CPU has 32KB of level 1 (L1) instruction cache, 32KB of L1 data cache, separate dedicated power domain, and includes one Neon™ and Vector Floating Point Unit (VFPv3) coprocessors. The dual Cortex-A9 MPU subsystem also includes standard CoreSight™ components to support SMP debug and emulation, snoop control unit (SCU), interrupt controller (GIC), clock and reset manager, Memory Adapter (MA) and Cache Management Unit (CMU).

The MPU subsystem handles transactions among the ARM® core, L3 interconnect EMIF controller (through the MA), L4-ABE, and interrupt controller (INTC). Hereafter in this chapter, references to the dual Cortex-A9 MPU subsystem, Cortex-A9 MPU subsystem, and MPU subsystem are equivalent.

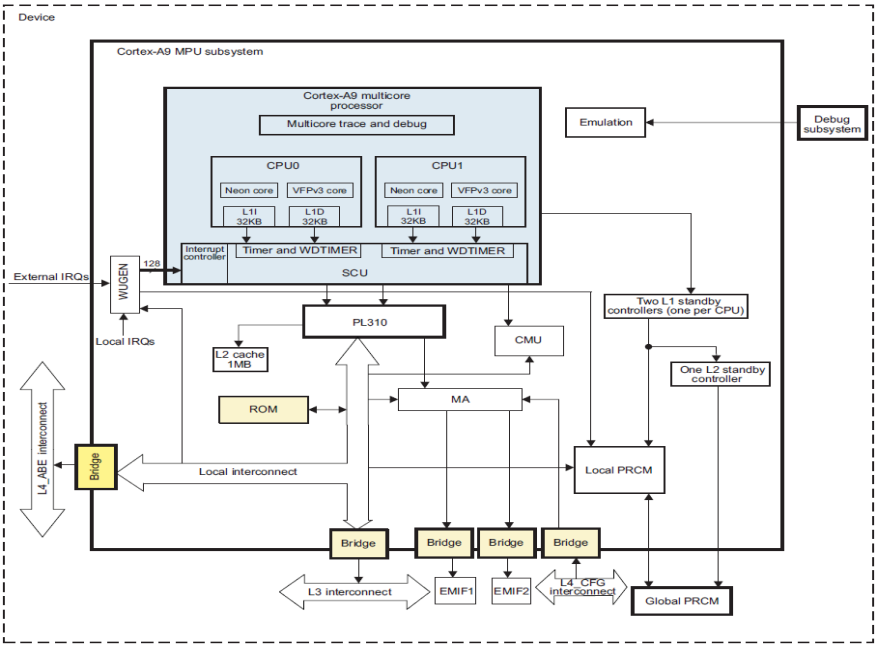

Figure 1: Block diagram of the MPU subsystem

3. *General-Purpose Interface (GPIO) Overview*

The general-purpose interface combines six general-purpose input/output (GPIO) banks. Each GPIO module provides 32 dedicated general-purpose pins with input and output capabilities; thus, the general-purpose interface supports up to 192 (6 × 32) pins. These pins can be configured for the following applications:

• Data input (capture)/output (drive)

• Keyboard interface with a debounce cell

• Interrupt generation in active mode upon the detection of external events. Detected events are processed by two parallel independent interrupt-generation submodules to support biprocessor operations.

• Wake-up request generation in idle mode upon the detection of external events

These modules do not include pad control (pullup/pulldown control, open-drain feature).

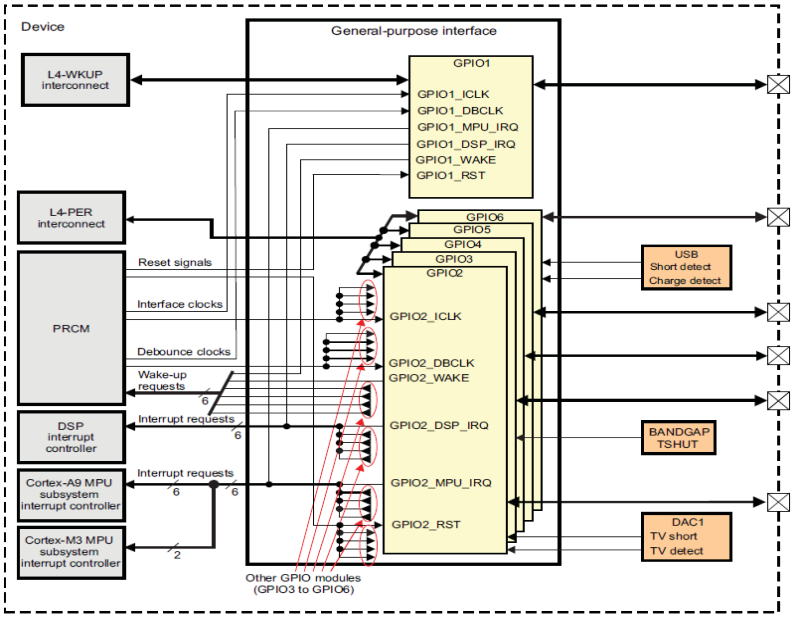

Figure 3: is an overview of the general-purpose interface.

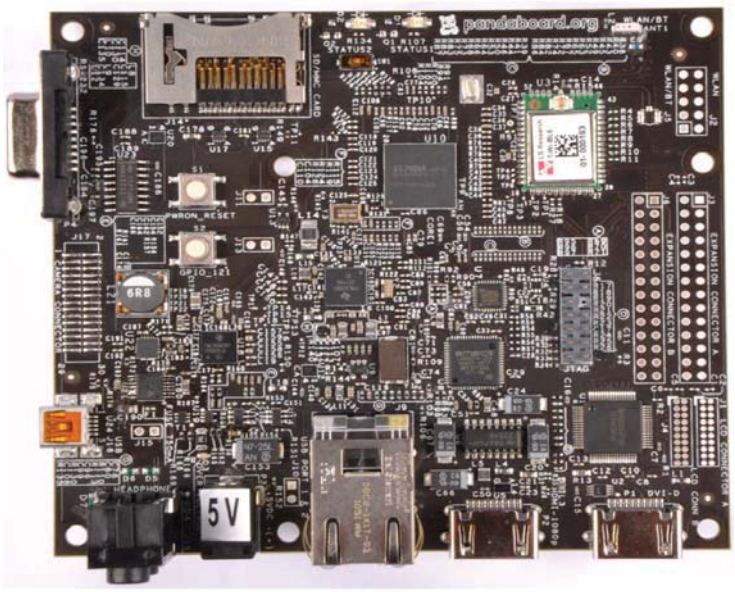

Figure 4: OMAP4460 board.

4. Frame work preparation for GPIO of OMAP4460 in kernel space

Step 1: Study GPIO Basics & OMAP4460 GPIO Controller block diagram, specifications & register programming model.
Step 2: Identify GPIO Device Driver Source code in kernel.
$ grep -i GPIO arch/arm/configs/omap2plus_defconfig

CONFIG_GPIOLIB=y
drivers/gpio/gpiolib.c

CONFIG_KEYBOARD_GPIO=y
drivers/input/keyboard/Makefile:obj-$(CONFIG_KEYBOARD_GPIO) += gpio_keys.o
drivers/input/keyboard/gpio_keys.c

CONFIG_GPIO_DEVRES=y
drivers/gpio/devres.c

CONFIG_OF_GPIO=y
drivers/gpio/gpiolib-of.c

CONFIG_DEBUG_GPIO=y
drivers/gpio/Makefile:ccflags-$(CONFIG_DEBUG_GPIO) += -DDEBUG
CONFIG_GPIO_SYSFS=y
drivers/gpio/gpiolib.c:#ifdef CONFIG_GPIO_SYSFS
drivers/gpio/gpiolib.c:#ifdef CONFIG_GPIO_SYSFS
drivers/gpio/gpiolib.c:#endif /* CONFIG_GPIO_SYSFS */

CONFIG_GPIO_TWL4030=y
drivers/gpio/Makefile:obj-$(CONFIG_GPIO_TWL4030) += gpio-twl4030.o

CONFIG_LEDS_GPIO=y
drivers/leds/Makefile:obj-$(CONFIG_LEDS_GPIO) += leds-gpio.o

CONFIG_LEDS_TRIGGERS=y
drivers/leds/Makefile:obj-$(CONFIG_LEDS_TRIGGERS) += led-triggers.o

CONFIG_GENERIC_GPIO=y
include/linux/gpio.h:#ifdef CONFIG_GENERIC_GPIO
include/linux/gpio.h:#else /* ! CONFIG_GENERIC_GPIO */
include/linux/gpio.h:#endif /* ! CONFIG_GENERIC_GPIO */

Step3: Prepare a Framework.

Step3a: GPIO Initialization (Bottom to Top) - Communication between Board specific layer to Low level Device Driver. Enable printf() statements in board specific file and low level device driver inside each and every function.

Board Specific Code:
---
1. Board Specific file: arch/arm/mach-omap2/board-omap4panda.c (board specific initalization - MUX,timers,clocks)
2. arch/arm/mach-omap2/gpio.c (gpio initalization, GPIO platform device registration happend with name "omap_gpio")

```
   omap2_gpio_dev_init ()
   {
     char *name = "omap_gpio";
   }
  arch/arm/plat-omap/include/plat/gpio.h
struct omap_gpio_platform_data {
      strutct omap_gpio_reg_offs *regs
}
```

arch/arm/mach-omap2/omap_hwmod.c (generic h/w module initalization)
arch/arm/mach-omap2/omap_hwmod_44xx_data.c (All platform data available here)

Low Level Device Driver:
---
1. drivers/gpio/gpio-omap.c

```
init ()
{
platform_driver_register(&omap_gpio_driver);
}
static struct platform_driver omap_gpio_driver = {
      .probe         = omap_gpio_probe,
      .driver        = {
            .name   = "omap_gpio",
            .pm     = &gpio_pm_ops,
            .of_match_table = of_match_ptr(omap_gpio_match),
      },
};
```

Base Address:
0x4A31 0000
0x4805 5000

0x4805 7000
0x4805 9000
0x4805 B000
0x4805 D000

II. CONCLUSION

The **GPIO (General Purpose Input Output)** are pins on the microprocessor to perform operations of input-output programmable power. Each pin can be assigned as input or output by programming easily and used to communicate with external devices.

REFERENCES

[1] SWPU235Z–February 2011–Revised April 2013

[2] IOSR Journal of Electronics and Communication Engineering(IOSR-JECE) ISSN: 2278-2834, ISBN: 2278-8735, PP: 32-36 www.iosrjournals.org Second International Conference on Emerging Trends in Engineering (SICETE) 32 | Page Dr.J.J.Magdum College of Engineering, Jaysingpur Image Processing Based on Embedded Linux 1Mr. S. M. Gramopadhye, 2Prof. R. T. Patil, 3Mr. A. N. Magdum, 4Mr. R. A. Chaugule1,2(RIT Sakharale, 3SGI Atigre, 4JJMCOE Jaysingpure).

[3] https://developer.mozilla.org/en US/docs/Mozilla/Firefox_OS/Pandaboard

[4] "OpenBSD/armv7". Retrieved 2014-05-15.

[5] http://lists.freebsd.org/pipermail/freebsd-hackers/2012-August/040263.html

[6] "Release notes for the Genode OS Framework 12.05". Retrieved 2016-10-28.

[7] Lee, Jeffrey (2011-08-02). "Have I Got Old News For You". The Icon Bar. RetrievedSeptember 28, 2011. […] Willi Theiss has recently announced that he's been working on a port of RISC OS to the PandaBoard […]

[8] "TI OMAP 4430 Panda / 4460 Panda ES Board Support Package". Retrieved2014-05-14.